ASP.NET Core 2

Guida completa per lo sviluppatore

Daniele Bochicchio, Cristian Civera
Moreno Gentili, Stefano Mostarda, Matteo Tumiati

aspitalia.com
The ASP.NET Community

HOEPLI
INFORMATICA

ASP.NET Core 2

Daniele Bochicchio Cristian Civera Moreno Gentili Stefano Mostarda Matteo Tumiati

ASP.NET Core 2

Guida completa per lo sviluppatore

EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2018
via Hoepli 5, 20121 Milano (Italy)
tel. +39 02 864871 – fax +39 02 8052886
e-mail hoepli@hoepli.it

www.hoepli.it

© Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.
Le fotocopie effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Centro Licenze e Autorizzazioni per le Riproduzioni Editoriali, Corso di Porta Romana 108, 20122 Milano, e-mail autorizzazioni@clearedi.org e sito web www.clearedi.org.

ISBN eBook 978-88-203-8673-3

Progetto editoriale e realizzazione: Maurizio Vedovati - Servizi editoriali (info@iltrio.it)
© Copertina: Sara Taglialegne

☺ Impaginazione: Vincenzo

© Realizzazione digitale: Promedia

Sommario

CONTENUTI DEL LIBRO **10**

A CHI SI RIVOLGE QUESTO LIBRO 10
CONVENZIONI 10
MATERIALE DI SUPPORTO ED ESEMPI 11
REQUISITI SOFTWARE PER GLI ESEMPI 11
CONTATTI CON L'EDITORE 11
CONTATTI, DOMANDE AGLI AUTORI 11

ASPITALIA.COM NETWORK **12**

CAPITOLO 1 - PRIMI PASSI CON .NET CORE E ASP.NET CORE **13**

GENESI DI .NET CORE 13
ANALISI DI .NET CORE 14
Creare applicazioni per .NET Core *15*
ASP.NET CORE 15
ASP.NET Core e .NET Framework *16*
.NET STANDARD 16
SCEGLIERE .NET CORE O .NET FRAMEWORK 17
CONCLUSIONI 18

CAPITOLO 2 - USARE .NET CORE CLI, VISUAL STUDIO E VISUAL STUDIO CODE PER INIZIARE CON ASP.NET **19**

SCEGLIERE L'AMBIENTE DI SVILUPPO 19
IDE o editor? *19*
INSTALLARE IL .NET CORE SDK 20
GESTIRE PROGETTI ASP.NET CORE CON IL .NET CORE CLI 21
Guida all'utilizzo del comando dotnet *21*
SELEZIONARE UNA VERSIONE DEL .NET CORE SDK 22
Elencare le SDK installate e determinare quella in uso *22*
Cambiare la versione del .NET Core SDK con il global.json *23*
IL PRIMO PROGETTO CON ASP.NET CORE 23
Scegliere un template di progetto *23*
Creare un progetto dagli ambienti di sviluppo *25*
Installare altri template di progetto *26*
COMPILARE ED ESEGUIRE L'APPLICAZIONE 26
Le applicazioni ASP.NET Core sono applicazioni console *28*
DEBUG DELL'APPLICAZIONE 28
Debug con Visual Studio *29*
Debug con Visual Studio Code *30*
Debug con Visual Studio for Mac *32*
CREARE UNA SOLUTION CON MOLTEPLICI PROGETTI 32
Creare la solution *33*
Aggiungere un progetto di tipo Class Library *33*
Aggiungere un progetto di unit testing *33*
LAVORARE CON I PACCHETTI NUGET 34
Aggiungere pacchetti NuGet al progetto *34*
Creare un nostro personale pacchetto NuGet *34*
PREPARARE L'APPLICAZIONE PER LA DISTRIBUZIONE 34
Il comando dotnet publish *35*
Cross-compilazione e pubblicazione self-contained *35*
Creare uno store a livello di sistema dei pacchetti NuGet *36*
Avviare l'applicazione nella macchina server *36*
CONCLUSIONI 37

CAPITOLO 3 - ANATOMIA DI UN PROGETTO ASP.NET CORE **38**

UNA VISIONE D'INSIEME DEL PROGETTO ASP.NET CORE 38
LE DIRECTORY DEL PROGETTO 38
La directory wwwroot *38*
Organizzare il codice in directory *39*
Le directory obj e bin *39*
LA CLASSE PROGRAM 40
Personalizzare il web host *41*

Indicare gli endpoint ... *41*
Scegliere un web server: Kestrel o HTTP.sys? ... *42*
Cambiare il percorso delle directory principali ... *44*
Impostare un environment ... *44*
Configurare il web host da fonti esterne ... *45*
Scegliere la classe di configurazione dell'applicazione ... *46*
CONFIGURARE L'APPLICAZIONE CON LA CLASSE STARTUP ... 46
USARE MIDDLEWARE DAL METODO CONFIGURE ... 47
Come funzionano i middleware ... *48*
AGGIUNGERE SERVIZI DAL METODO CONFIGURESERVICES ... 48
Cos'è un servizio ... *49*
CONFIGURARE L'APPLICAZIONE ... 49
Variare la configurazione in base all'environment ... *49*
Leggere la configurazione in maniera fortemente tipizzata ... *50*
Che fine hanno fatto il web.config e il global.asax? ... *51*
IL FILE .CSPROJ ... 52
Il metapacchetto Microsoft.AspNetCore.App ... *52*
Il tool Microsoft.VisualStudio.Web.CodeGeneration.Tools ... *53*
CONCLUSIONI ... 53

CAPITOLO 4 - PRIMI PASSI CON ASP.NET CORE MVC ... 54

IL PATTERN MODEL-VIEW-CONTROLLER ... 54
I SERVIZI DI ROUTING ... 55
IL CONTROLLER E IL MODEL ... 55
LA VIEW E GLI HTML HELPER ... 57
Gestire una form di input ... *60*
ASP.NET CORE MVC E PROGETTI COMPLESSI: LE AREE ... 61
CONCLUSIONI ... 64

CAPITOLO 5 - CONTROLLER E ROUTING ... 65

URL ROUTING IN ASP.NET CORE MVC ... 65
Constraint sulle route ... *66*
ANATOMIA DI UN CONTROLLER ... 67
Proprietà e metodi di supporto della classe Controller ... *67*
Ciclo di vita di un controller ... *68*
USO DELLA DEPENDENCY INJECTION NEI CONTROLLER ... 69
Utilizzare un motore di IoC container differente ... *71*
LA ACTION COME GESTORE DELLA RICHIESTA ... 71
L'interfaccia IActionResult e i diversi tipi di risposta ... *72*
Controllo dell'esecuzione di una action ... *75*
Esecuzione asincrona di una action ... *76*
L'INFRASTRUTTURA DEI FILTRI ... 77
CONCLUSIONI ... 78

CAPITOLO 6 - LE VIEW E RAZOR ... 79

CREARE VIEW IN ASP.NET CORE MVC GRAZIE A RAZOR ... 79
LA SINTASSI DI BASE ... 80
Branch e cicli ... *80*
Definire funzioni in una view ... *81*
LE VIEW E IL MODEL: TIPIZZAZIONE DEBOLE E FORTE ... 81
CONSISTENZA GRAFICA TRA LE PAGINE: LA LAYOUT VIEW ... 85
Sfruttare le layout view nel progetto ... *85*
La view _ViewStart ... *86*
Definire sezioni aggiuntive in una layout view ... *87*
SEMPLIFICARE IL CODICE DELLE VIEW: GLI HTML E I TAG HELPER ... 87
Gli HTML Helper ... *88*
Il metodo RouteLink ... *89*
Il metodo Raw ... *89*
Il metodo Partial e le partial view ... *89*
I TAG HELPER E LA GESTIONE DEL MARKUP ... 90
Il Tag helper Partial ... *91*
Il Tag helper Environment ... *91*
Il Tag helper Anchor ... *91*

Il Tag helper Image *92*
CONCLUSIONI 92

CAPITOLO 7 - GESTIONE DELLE FORM 93

GENERARE LA FORM DA UN MODEL 93
Definire il model *93*
Definire il controller e la action *95*
Creare la view *95*
VALIDARE L'INPUT DELL'UTENTE 100
Gli attributi di validazione *100*
Applicare la validazione sulla view *102*
Personalizzare la validazione *104*
Gestire gli errori nella action *107*
IL MODEL BINDER 107
CONCLUSIONI 108

CAPITOLO 8 - LA GESTIONE DELLO STATO 109

COME FUNZIONA UNA RICHIESTA HTTP? 109
SCENARI DI GESTIONE DELLO STATO 109
LO STATO CON I CAMPI HIDDEN 110
LO STATO ATTRAVERSO I COOKIE 110
GESTIONE DELLO STATO NELLA SESSIONE 112
PASSARE VALORI TRAMITE QUERYSTRING 114
GESTIONE DEI DATI TEMPORANEI CON TEMPDATA 114
MANTENERE DATI DI UNA RICHIESTA: HTTPCONTEXT.ITEMS 115
UTILIZZARE LA CACHE 116
Local cache *116*
Distributed cache *119*
HTML CACHING 120
RESPONSE CACHING 121
CONCLUSIONI 122

CAPITOLO 9 - ACCEDERE AI DATI CON ADO.NET 123

ARCHITETTURA DI ADO.NET 123
Il namespace System.Data.Common *123*
Data provider *124*
Il namespace System.Data.SqlTypes *124*
Il namespace System.Data *124*
LAVORARE CON ADO.NET 125
Stabilire una connessione *125*
Esecuzione di un comando *127*
Scrivere dati in transazione *129*
Lettura del risultato di una query *131*
Modalità disconnessa in ADO.NET *132*
Ricercare dati in un DataSet *133*
CONCLUSIONI 133

CAPITOLO 10 - ACCEDERE AI DATI CON ENTITY FRAMEWORK CORE 134

COSA È UN O/RM 134
MAPPARE IL MODELLO A OGGETTI SUL DATABASE 135
Disegnare le classi *135*
Creare il contesto *137*
Mapping tramite convenzioni *137*
Mapping tramite API *138*
Mapping tramite data annotation *141*
Generare automaticamente le classi dal database *142*
Configurare Entity Framework Core con ASP.NET *143*
RECUPERARE I DATI DAL DATABASE 143
Ottimizzare il fetching *145*
Lazy loading *145*
Querying avanzato *146*
SALVARE I DATI SUL DATABASE 147
Persistere un nuovo oggetto *148*

Persistere le modifiche a un oggetto ... *148*
Cancellare un oggetto dal database ... *149*
Manipolare gli oggetti nel change tracker ... *150*
Funzionalità aggiuntive di Entity Framework Core ... 150
Conclusioni ... 150

CAPITOLO 11 - SVILUPPARE SERVIZI RESTFUL CON ASP.NET CORE WEBAPI ... 152

I servizi RESTful ... 152
Implementare un endpoint per le API con ASP.NET Core ... 153
Status code e gestione degli errori ... 157
Formatter e content negotiation ... 159
Lavorare con le collection ... 162
Eseguire aggiornamenti ... 163
Hypermedia As The Engine Of Application State (HATEOAS) ... 165
Conclusioni ... 168

CAPITOLO 12 - SVILUPPARE SERVIZI RESTFUL AVANZATI CON ASP.NET CORE WEBAPI, WEBHOOK E SIGNALR .. 169

Documentazione con Swagger ... 169
Gestione delle versioni ... 172
Gestire endpoint basati su WebHook ... 174
Comunicazione in tempo reale con SignalR ... 175
Conclusioni ... 180

CAPITOLO 13 - GESTIONE E DIAGNOSTICA DEGLI ERRORI ... 182

La gestione degli errori a runtime ... 182
Gli errori durante lo sviluppo ... 182
Gli errori per l'utente ... *184*
Le pagine per gli status code ... 185
Tracciare gli eventi ... 187
Configurare il logging ... 189
Un provider per il logging personalizzato ... 190
L'integrazione con Azure App Service ... 192
Conclusioni ... 193

CAPITOLO 14 - ESTENDERE ASP.NET CORE ... 194

Personalizzare l'host di esecuzione ... 194
Iniettare l'utilizzo di una libreria ... *194*
Configurazione separata dell'app ... *195*
Avviare servizi in background ... *195*
Creare librerie con view Razor ... 197
Caricare a runtime parti dell'applicazione ... 198
I middleware e il contesto HTTP ... 199
Creare un middleware personalizzato ... 200
Creare un middleware come componente ... 202
Concetti avanzati sui middleware ... 203
Applicare filtri ai controller MVC ... 204
Creare un filtro personalizzato ... 206
Utilizzare la dependency injection con i filtri ... 207
Conclusioni ... 208

CAPITOLO 15 - MODEL BINDER E TAG HELPER PERSONALIZZATI ... 210

Il model binding ... 210
Creazione di un model binder personalizzato ... *212*
Model binding di interfacce ... 215
Modifica dei metadati ... 217
Sviluppare un tag helper personalizzato ... 219
Esporre proprietà sul tag helper ... *220*
Generare HTML nei tag helper ... *221*
Conclusioni ... 222

CAPITOLO 16 - SUPPORTO ALLA GLOBALIZZAZIONE E INTERNAZIONALIZZAZIONE ... 223

I requisiti minimi ... 223
Internazionalizzazione, globalizzazione e localizzazione ... 223

LAVORARE CON LE CULTURE 224
UTILIZZARE I FILE DI RISORSE 227
LA LOCALIZZAZIONE DEI CONTENUTI 230
IL LOCALIZATION MIDDLEWARE 232
LA LOCALIZZAZIONE DELLE VIEW 235
LE DATA ANNOTATION 237
CONCLUSIONI 238

CAPITOLO 17 - AUTENTICAZIONE, AUTORIZZAZIONE E IDENTITÀ 239

ACCEDERE A UN'APPLICAZIONE 239
La fase di login *240*
Un'identità formata da claim *240*
La fase di autenticazione *240*
La fase di autorizzazione *241*
SCEGLIERE UN SISTEMA DI MEMBERSHIP 242
Membership con ASP.NET Core Identity *242*
Membership con Azure Active Directory B2C *243*
CONFIGURARE L'AUTENTICAZIONE: ACCOUNT LOCALI 244
Autenticazione su cookie con ASP.NET Core Identity *244*
Autenticazione su cookie senza ASP.NET Core Identity *245*
Autenticazione su token JWT *247*
CONFIGURARE L'AUTENTICAZIONE: IDENTITÀ FEDERATA 248
Login con i social network *248*
Login su provider esterni con OAuth 2.0 e OpenID Connect *249*
Autenticazione Windows *249*
Autenticazione con Azure Active Directory *250*
AUTORIZZAZIONE IN ASP.NET CORE MVC E WEB API 251
L'attributo AuthorizeAttribute *251*
L'attributo AllowAnonymousAttribute *252*
Autorizzazione in base al ruolo *252*
Usare le policy per criteri avanzati di autorizzazione *252*
Policy con logica di autorizzazione personalizzata *253*
Autorizzazione in base all'authentication scheme *254*
Autorizzazione on demand con IAutorizationService *255*
CONCLUSIONI 255

CAPITOLO 18 - SICUREZZA NELLE APPLICAZIONI ASP.NET CORE 256

PERSONALIZZARE L'ACCOUNT DELL'UTENTE 256
Aggiungere proprietà personalizzate al profilo dell'utente *256*
Personalizzare la pagina di registrazione *257*
SICUREZZA DEGLI ACCOUNT LOCALI 258
Criteri di sicurezza della password *258*
Chiedere la conferma di registrazione *259*
Reimpostazione della password *260*
Two-factor authentication *260*
Protezione dai tentativi di accesso brute force *261*
Sign-out remoto *262*
CONFORMITÀ CON LE NORMATIVE 262
Controllo dei dati da parte dell'utente per il GDPR *262*
Rispetto della normativa europea sui cookie *263*
GESTIRE GLI UTENTI CON ASP.NET CORE IDENTITY 263
Creare un utente programmaticamente *264*
Interrogare il database degli utenti locali *264*
Assegnare claim e modificare i dati di login dell'utente *265*
PROTEGGERSI DAGLI ATTACCHI 265
Proteggere l'applicazione con un certificato SSL *265*
Forzare il traffico su HTTPS *267*
Cross Site Scripting (XSS) *267*
Cross Site Request Forgery (CSRF o XSRF) *268*
Open Redirect *269*
Cross-Origin Resource Sharing (CORS) *269*
CONFIGURARE L'APPLICAZIONE IN MANIERA SICURA 270
User secrets *270*
Azure Key Vault *271*

PROTEGGERE I DATI SENSIBILI 272
Usare Data Protection API per cifrare dati *272*
Gestire le chiavi crittografiche di Data Protection API *273*
CONCLUSIONI 275

CAPITOLO 19 - UTILIZZARE JAVASCRIPT IN APPLICAZIONI ASP.NET 276

GESTIRE LE LIBRERIE JAVASCRIPT 276
JQUERY 277
Effettuare ricerche nel DOM *278*
Manipolare gli oggetti *278*
Gestire gli eventi *279*
Effettuare chiamate AJAX *279*
jQueryUI *280*
BOOTSTRAP 281
VUE.JS 283
CONCLUSIONI 286

CAPITOLO 20 - GLI STRUMENTI DI VISUAL STUDIO PER LO SVILUPPO WEB 287

BUNDLING E MINIFICATION 287
Minification *287*
Bundling *288*
Integrazione con ASP.NET Core *288*
LINK E SCRIPT TAG HELPER 288
Utilizzare NPM *289*
Utilizzare Gulp *291*
UTILIZZARE GRUNT 292
CONCLUSIONI 294

CAPITOLO 21 - SINGLE PAGE APPLICATION E ASP.NET CORE 295

INTRODUZIONE ALLE SINGLE PAGE APPLICATION 295
ANGULAR 296
REACTJS 302
CONCLUSIONI 304

CAPITOLO 22 - CONFIGURARE E PUBBLICARE UN'APPLICAZIONE ASP.NET CORE 306

MODIFICARE IL COMPORTAMENTO SECONDO L'AMBIENTE DI DESTINAZIONE 306
PUBBLICAZIONE E HOSTING 310
Configurazione di NGINX *311*
Configurazione di Apache HTTP Server *314*
Configurazione di IIS *315*
PUBBLICAZIONE CON MICROSOFT AZURE 318
PUBBLICAZIONE CON DOCKER 322
CONCLUSIONI 326

INFORMAZIONI SUL LIBRO 327

Contenuti del libro

Questo libro affronta tutte le principali problematiche che ogni giorno, come sviluppatori, vi troverete a dover affrontare nella realizzazione di applicazioni web con ASP.NET Core. Scritto da quattro esperti del settore, membri dello staff di ASPItalia.com, la più grande community italiana dedicata allo sviluppo web, il libro è stato pensato per mostrarvi tutte le possibilità offerte da ASP.NET Core.

ASP.NET Core 2 – Guida completa per lo sviluppatore si propone di guidarvi attraverso un percorso formativo graduale e completo, che include argomenti come Razor, LINQ, l'accesso ai dati, la creazione di UI avanzate, la protezione e la progettazione delle applicazioni web, utilizzando uno stile chiaro e ricco di esempi, scritti in C#.

Il libro si suddivide in sei parti, ciascuna delle quali risponde a un insieme di esigenze di sviluppo specifiche.

Le informazioni di base, riguardanti gli strumenti per iniziare a sviluppare con ASP.NET Core compongono la prima parte, che include i primi 3 capitoli.

La seconda parte, dal capitolo 4 al capitolo 8, è dedicata ad analizzare le caratteristiche di base di ASP. NET Core, partendo dall'ambiente, per quanto concerne la pagina e le sue caratteristiche di base, ovvero ASP.NET Core MVC e Razor, fino ad arrivare alla gestione di form e stato.

L'accesso ai dati e lo sfruttamento dei servizi sono invece gli argomenti della terza parte del libro, che approfondisce tutti gli aspetti legati all'estrazione e visualizzazione dei dati.

La quarta parte, dal capitolo 13 al capitolo 16, comprende una serie di capitoli completamente dedicati ai concetti avanzati, che consentono di estendere il funzionamento di ASP.NET Core.

La quinta parte, che prendere i capitoli 17 e 18, si concentra sugli aspetti legati alla sicurezza, analizzando i meccanismi che consentono di proteggere parti dell'applicazione e di scrivere codice sicuro.

Infine, la sesta parte tratta dell'integrazione della parte server side con quella client side, analizzando l'integrazione con JavaScript, come consumare servizi e, infine, come distribuire l'applicazione.

A chi si rivolge questo libro

L'idea che sta alla base della scrittura di questo libro è quella di fornire un rapido accesso alle informazioni principali che caratterizzano la versione 2.1 di ASP.NET Core e .NET Core. Quando presenti, le novità rispetto alle versioni precedenti sono messe in risalto, ma questo libro è indicato anche per chi è digiuno di ASP.NET o ASP.NET Core e desidera imparare l'uso di questa tecnologia partendo da zero. Conoscendo già ASP.NET WebForms o ASP.NET MVC, i primi capitoli potrebbero contenere un ripasso di concetti già conosciuti. Suggeriamo comunque di leggerli, poiché alcune piccole differenze tra ASP.NET Core e ASP.NET sono sempre presenti.

Al momento di stampare questo libro, la versione 2.2 è stata rilasciata come preview. Poiché ASP.NET Core ha una roadmap pubblica, possiamo già essere certi che si tratta di una versione che non aggiunge funzionalità radicalmente differenti da quanto disponibile. Per questo motivo, tutti i contenuti presenti nel libro sono di fatto già pronti per questa release.

Questo libro non contiene una trattazione dei linguaggi, per l'approfondimento dei quali vi consigliamo la valutazione di altri libri, sempre appartenenti a questa stessa collana:

C# 5 – Guida completa per lo sviluppatore ISBN: 978-88-203-5253-0
www.hoeplieditore.it/5253-0

Per comprendere appieno molti degli esempi e degli ambiti in cui vi dovrete muovere all'interno del libro, è richiesta da parte del lettore una buona conoscenza del linguaggio HTML, per meglio comprendere e apprezzare il modello dichiarativo proprio di ASP.NET Core. La conoscenza di JavaScript e CSS può aiutare a comprendere al meglio alcune parti. Per approfondire queste tematiche consigliamo la lettura di questo libro:

HTML5 con CSS e JavaScript, 2ª edizione ISBN: 978-88-203-8525-5
www.hoeplieditore.it/8525-5

Convenzioni

All'interno di questo volume abbiamo utilizzato stili differenti secondo il significato del testo, così da rendere più netta la distinzione tra tipologie di contenuti differenti.

I termini importanti sono spesso indicati in grassetto, così da essere più facilmente riconoscibili.

Il testo contenuto nelle note è scritto in questo formato. Le note contengono informazioni aggiuntive relativamente a un argomento o ad aspetti particolari ai quali vogliamo dare una certa rilevanza.

Gli esempi contenenti codice o markup sono rappresentati secondo lo schema riportato di seguito. Ciascun esempio è numerato in modo da poter essere referenziato più facilmente nel testo e recuperato dagli esempi a corredo.

Esempio 1.1 - Linguaggio

```
Codice
Codice importante, su cui si vuole porre l'accento
Altro codice
```

Per `namespace`, `classi`, `proprietà`, `metodi` ed `eventi` è utilizzato `questo stile`. Qualora vogliamo attirare la vostra attenzione su uno di questi elementi, per esempio perché è la prima volta che viene menzionato, lo stile che troverete sarà `questo`.

Materiale di supporto ed esempi

Allegata a questo libro è presente una nutrita quantità di esempi, che riprendono sia gli argomenti trattati sia quelli non approfonditi. Il codice può essere scaricato agli indirizzi www.hoeplieditore.it/7290-3 e http://books.aspitalia.com/ASP.NET-Core/, dove saranno anche disponibili gli aggiornamenti e tutto il materiale collegato al libro.

Requisiti software per gli esempi

Questo è un libro dedicato ad ASP.NET Core 2.1, con un particolare riferimento alla tecnologia in quanto tale, pertanto non è necessario nient'altro che il .NET Core SDK in versione 2.1 (o successiva).

Ove si eccettuino pochi casi particolari, comunque evidenziati, per visionare e testare gli esempi potete utilizzare una qualsiasi versione di Visual Studio 2017 (o successivo) (per Windows o macOS), oppure Visual Studio Code (su Windows, Linux e macos). Visual Studio Code è scaricabile gratuitamente senza limitazioni particolari e utilizzabile liberamente, anche per sviluppare applicazioni a fini commerciali, all'indirizzo http://code.visualstudio.com/.

Per quanto concerne l'accesso ai dati, nel libro facciamo riferimento principalmente a SQL Server. Vi raccomandiamo di utilizzare la versione Express di SQL Server, liberamente scaricabile all'indirizzo http://www.microsoft.com/express/sql/. Il tool per gestire questa versione si chiama SQL Server Management Tool Express, ed è disponibile allo stesso indirizzo.

Contatti con l'editore

Per qualsiasi necessità, potete contattare direttamente l'editore attraverso il sito

http://www.hoeplieditore.it/

Contatti, domande agli autori

Per rendere più agevole il contatto con gli autori, abbiamo predisposto un forum specifico, raggiungibile all'indirizzo http://forum.aspitalia.com/, in cui saremo a vostra disposizione per chiarimenti, approfondimenti e domande legate al libro.

Potete partecipare, previa registrazione gratuita, alla community di ASPItalia.com Network.

Vi aspettiamo!

ASPItalia.com Network

ASPItalia.com Network, nata dalla passione dello staff per la tecnologia, è supportata da oltre vent'anni di esperienza con ASPItalia.com per garantirvi lo stesso livello di approfondimento, aggiornamento e qualità dei contenuti su tutte le tecnologie di sviluppo del mondo Microsoft. Con oltre 70.000 iscritti alla community, i forum rappresentano il miglior luogo in cui porre le vostre domande riguardanti tutti gli argomenti trattati!

ASPItalia.com si occupa principalmente di tecnologie dedicate al Web, da ASP.NET in poi, con un'aggiornata e nutrita serie di contenuti pubblicati nei precedenti anni di attività, che spaziano da ASP a Windows Server, passando per security e XML.

Il network comprende:

- HTML5Italia.com con HTML5, CSS3, ECMAScript 5 e tutto quello che ruota intorno agli standard web per costruire applicazioni che sfruttino al massimo il client e le specifiche web.

- LINQItalia.com, con le sue pubblicazioni, approfondisce tutti gli aspetti di LINQ, passando per i vari flavour LINQ to SQL, LINQ to Objects, LINQ to XML oltre a Entity Framework.

- WindowsAzureItalia.com pubblica script, risorse, tutorial e articoli dedicati alle tecnologie cloud di casa Microsoft.

- WinFXItalia.com, in cui sono presenti contenuti sullo sviluppo con il .NET Framework e i relativi linguaggi.

- WinRTItalia.com tratta gli aspetti legati alla creazione di applicazioni per Windows 10, dall'UX fino allo sviluppo.

1
Primi passi con .NET Core e ASP.NET Core

Il .NET Framework è stato rilasciato nel lontano 2002 e, da allora, il mondo del software si è molto evoluto. ASP.NET, con esso, ha subito profonde rivoluzioni, che l'hanno portato a diventare uno dei framework per lo sviluppo di applicazioni web più utilizzati tra quelli in circolazione.

Quasi vent'anni, nel campo IT, sono più o meno cinque evoluzioni tecnologiche significative: difatti, il .NET Framework è passato attraverso la diffusione del desktop, del web, dei palmari, delle RIA, degli smartphone e, di nuovo, per la rinascita del web, con in mezzo l'evoluzione che ci ha portati al cloud. Durante queste evoluzioni, ASP.NET, ha subito numerosi cambiamenti e aggiornamenti, per lo più legati alle novità introdotte nel linguaggio C# (e VB) o alla comparsa di nuove tecnologie e paradigmi (come LINQ, AJAX, il cloud computing ecc.). Inoltre, all'iniziale approccio per creare l'interfaccia, basato su Web Forms, è stata affiancata negli anni un'alternativa, che implementa il pattern MVC (Model-View-Controller) e consente di aggiungere ulteriori frecce al nostro arco di sviluppatori web.

Pur essendo fondamentalmente sempre in movimento, il .NET Framework è stato (e sempre resterà) storicamente legato a Windows. Oggi, però, la situazione è differente rispetto al 2002 e limitare un framework a una sola piattaforma (cosa che sembrava perfettamente logica vent'anni fa) è una scelta del tutto controproducente. Ormai, abbiamo server web all'interno di qualsiasi tipo di device, da quelli IOT (Internet Of Things), alle TV, ai termostati e, ovviamente, ai servizi web classici, per cui il tema di un nuovo framework capace di portare ASP.NET e C# verso nuovi orizzonti oggi è tutt'altro che campato per aria.

Dal successo e dalle premesse che hanno reso .NET Framework uno degli ambienti di sviluppo più apprezzati, è nato quindi .NET Core.

Nel corso di questo primo capitolo illustriamo le nozioni basilari riguardanti .NET Core, con particolare riferimento ai concetti che sono alla base di .NET Core e, quindi, di ASP.NET Core.

Genesi di .NET Core

.NET Core nasce dall'esigenza di evolvere in maniera moderna i concetti alla base del .NET Framework. Oggi, un framework moderno per il web deve essere cross-platform, per non precludere opportunità agli sviluppatori, e deve essere aggiornato di frequente.

Il .NET Framework, per sua natura, è sempre stato fortemente legato ai cicli di sviluppo di Visual Studio, seguendo un rilascio ogni due anni: due anni, nell'ambito web, possono tranquillamente essere considerati come un lasso temporale pari a dieci anni, poiché i trend sono molto rapidi a diventare realtà e il progresso è molto accelerato.

Microsoft, dunque, era di fronte a una scelta: provare a rendere .NET Framework cross-platform, con il rischio di rompere la compatibilità, oppure provare a evolvere i concetti alla base di .NET in una maniera nuova. Alla fine, ha prevalso questo orientamento: così è nato .NET Core.

Cos'è .NET Core? È una nuova implementazione dei concetti alla base di .NET, così come lo è .NET Framework. Se avete sviluppato con Silverlight, Xamarin o Universal Windows Platform, sapete che questi toolkit sono implementazioni di .NET create per risolvere problemi specifici, in ambiti in cui il .NET Framework sarebbe stato eccessivo. Sono, insomma, delle versioni di .NET pensate per risolvere problematiche specifiche, mentre .NET Framework è un framework più completo, frutto dell'insieme di alcuni toolkit (come ASP.NET, WPF, WinForms).

Volendo tirare le somme, .NET Core è solo l'ultima delle varianti di .NET, creata per consentirci di sviluppare applicazioni cross-platform per il web, basandoci sui concetti tipici di .NET Framework e di ASP.NET, nella sua variante chiamata ASP.NET Core.

> *.NET Core è un reboot di .NET. .NET Framework, infatti, non consente di re-ingegnerizzare tutti i pezzi di cui è composto per garantirne una portabilità cross-platform. Piuttosto che creare versioni di .NET Framework incompatibili tra loro, Microsoft ha preferito differenziarle totalmente, a partire dal nome stesso.*

Ogni implementazione di .NET porta con sé una serie di ideali e di servizi, che nient'altro sono che il supporto agli stessi linguaggi (C#, su cui ci soffermeremo in questo libro, ma anche Visual Basic), il runtime (CLR - Common Language Runtime – e JIT-ter - Just in Time compiler) e un po' di librerie condivise.

Se avete già sviluppato applicazioni per .NET Framework (o una delle sue varianti), passare a creare applicazioni per .NET Core non necessita di ulteriori conoscenze. Se, invece, state cercando di capire con questo libro se .NET Core fa per voi, è opportuno segnalare che i prerequisiti per uno sviluppatore di applicazioni per .NET Core sono i medesimi richiesti a uno sviluppatore .NET Framework e riguardano principalmente il linguaggio C#. Per questo motivo, non sono trattati direttamente in questo libro, ma vi consigliamo la lettura del nostro libro C# 2019 e .NET - Guida completa per lo sviluppatore, edito sempre da Hoepli.

La natura cross-platform di .NET Core è senza dubbio un vantaggio e non deve spaventare uno sviluppatore abituato a .NET Framework su Windows: come vedremo nel corso del prossimo capitolo, infatti, possiamo certamente optare per soluzioni cross-platform ma, se utilizziamo Windows, Visual Studio resta decisamente consigliabile.

Generalmente, nel corso di questo libro facciamo riferimento in maniera generica a Visual Studio, parlando della versione 2017 Update 15.7. Molti dei concetti e delle funzionalità sono validi anche per tutte le versioni successive. Al momento di scrivere questo libro, Visual Studio 2019 è stato solo annunciato.

.NET Core, insomma, è l'essenza di .NET, presa e portata a un livello più ampio. Come facciamo a capire se .NET Core faccia al caso nostro? .NET Core è la scelta ideale per chi vuole introdurre subito le innovazioni tecnologiche nelle proprie applicazioni web e in ambito cross plataform, grazie al suo ciclo di rilascio di update che è più o meno assestato su tre mesi, laddove .NET Framework è, invece, una scelta più conservativa, ideale per chi preferisce stabilità, poiché i rilasci sono molto meno frequenti (una volta ogni due anni).

Il prezzo da pagare, scegliendo .NET Framework, è quello di aspettare per avere le innovazioni, che prima sono introdotte all'interno di .NET Core (oppure di non riceverle affatto, poiché le stesse dipendono da componenti non facilmente portabili).

In estrema sintesi, .NET Core è la piattaforma di sviluppo che al momento riceve le maggiori attenzioni da parte di Microsoft ed è la scelta ideale per chi vuole creare applicazioni web moderne.

Grazie alla sua natura open source, poi, .NET Core è maggiormente allineato al modo di sviluppare moderno, con il codice sorgente sempre disponibile e la roadmap e il relativo sviluppo fatto in maniera open. Nonostante accetti pull request dalla community, Microsoft mantiene la governance del progetto e lo sviluppo dello stesso, così da garantire la qualità del prodotto finale e il normale ciclo di vita che ci si aspetta da una soluzione pensata anche per il mercato enteprise. Tutte le informazioni sono sempre disponibili su http://github.com/dotnet/.

Chi segue Microsoft sa quanto abbia iniziato a puntare sull'open source, con una trasformazione radicale del modo di sviluppare software e offrire servizi, tanto che, oggi, è l'azienda con il più alto numero di progetti open source su GitHub (il più famoso repository di software open source, che Microsoft ha anche acquistato), su cui è ospitato lo stesso codice sorgente di .NET Core e di tutti i prodotti correlati.

Analisi di .NET Core

Attualmente .NET Core, a differenza di .NET Framework, che è limitato solo a Windows, supporta direttamente questi sistemi operativi:

- Windows Client: 7, 8.1, 10 (dalla build 1607).
- Windows Server: 2008 R2 SP1 o successivo.
- macOS: 10.12 o successivo.
- RHEL: 7 o successivo.
- Fedora: 26 o successivo.
- openSUSE: 42.3 o successivo.
- Debian: 8 o successivo.
- Ubuntu: 14.04 o successivo.
- Alpine Linux: 3.6 o successivo.
- SLES: 12 o successivo.

Il supporto è disponibile per architetture Windows x86 e x64, macOS x64, Linux x64 e Linux ARM32 (il processore utilizzato da diversi dispositivi, tra cui Raspberry PI). Alcune piattaforme, pur non essendo supportate direttamente, lo sono grazie al lavoro fatto dalla community. Le piattaforme e i sistemi operativi direttamente supportati, però, godono del normale ciclo di supporto di Microsoft, con la possibilità, per le aziende, di acquistare anche i pacchetti di supporto ad hoc, che garantiscono aggiornamenti e supporto specifici. Pur essendo un prodotto open source, insomma, .NET Core gode dello stesso supporto di tutti gli altri prodotti Microsoft.

Tornando all'analisi delle piattaforme supportare, dobbiamo notare, in particolare, il supporto a ARM32, che apre la porta a tutta una serie di dispositivi (come le TV) che, a una prima analisi, mai penseremmo che possano beneficiare di un runtime per creare applicazioni web.

In realtà, ormai il web è oltre una serie di semplici pagine HTML ed è sempre più sfruttato per creare endpoint raggiungibili via HTTP, per offrire, per esempio, risultati in formato JSON.

Creare applicazioni per .NET Core

.NET Core consente principalmente di creare due tipologie di applicazioni:

- .NET Core Console Application
- ASP.NET Core Web Application

Le applicazioni di tipo console altro non sono che applicazioni da riga di comando: prendono in input quello che viene passato e producono un output. Le applicazioni basate su ASP.NET Core, di fatto, sono una specializzazione di queste, dove la console application lancia un server web che resta attivo e in ascolto delle richieste. In realtà, questo meccanismo (come vedremo nei prossimi capitoli) varia in funzione del sistema operativo e dell'ambiente di hosting, ma questa semplificazione ci consente di capire meglio come funziona ASP.NET Core. Di fatto, le applicazioni ASP.NET Core sono molto più simili alle applicazioni console di quanto lo siano mai state quelle ASP.NET.

In generale, ASP.NET Core è una parte di .NET Core specificamente pensata per creare applicazioni web. Attualmente è l'unico toolkit presente in .NET Core, che non ha il supporto per la creazione di UI native, a differenza di .NET Framework, che fornisce questi servizi grazie a WPF o WinForms, due toolkit creati per sviluppare applicazioni native per Windows. A partire da .NET Core 3, Microsoft inizierà a supportare un toolkit per creare applicazioni solo per Windows, offrendo il supporto a WPF e Universal Windows Platform; ma con l'attuale versione 2.1 questo non è ancora possibile.

La figura 1.1 mostra alcuni dei componenti chiave di.NET Core e li raffronta con il corrispondente in .NET Framework e Xamarin, piattaforma con cui condividono alcune funzionalità.

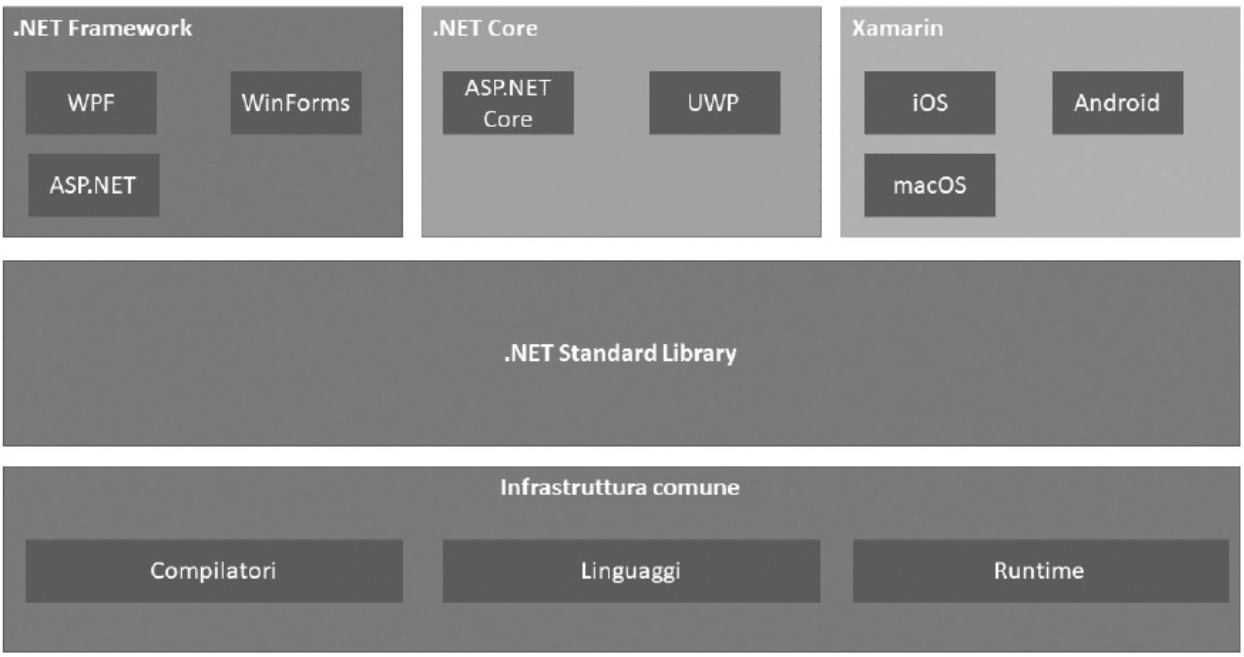


Figura 1.1 – I vari componenti di .NET Framework, .NET Core e Xamarin.

ASP.NET Core, a propria volta, è l'insieme delle funzionalità che ci consentono di creare applicazioni orientate al web. Cerchiamo di capire com'è fatto ASP.NET Core e quali servizi offra, prima di procedere oltre.

ASP.NET Core

ASP.NET Core è un insieme di servizi orientati a creare applicazioni web all'interno di .NET Core. In dettaglio, è composto da:

- un runtime, che si occupa di prendere in carico le richieste e generare le risposte;
- un server web embedded, chiamato Kestrel;
- uno strato di servizi per adattarsi ad altri server web o reverse proxy, come IIS o nginx.
- alcuni servizi orientati a creare pagine HTML, come MVC e Razor Pages;
- alcuni servizi orientati a creare endpoint di servizi REST, con Web API;
- un motore per applicazioni real-time, con SignalR;
- un insieme di servizi di sicurezza.

Tutti questi servizi si appoggiano al relativo SDK (Software Development Kit), che serve per creare le applicazioni e che offre alcuni servizi che introdurremo nel prossimo capitolo.

ASP.NET Core è, a propria volta, dotato di componenti che consentono di creare applicazioni che risolvano problematiche specifiche. In questo libro tratteremo specificamente MVC, Web API e SignalR, mentre non ci soffermeremo molto sulle Razor Pages: queste ultime sono infatti una semplificazione del pattern MVC e sono maggiormente indicate in progetti semplici, quando lo scopo è creare semplici siti web dinamici. Lo scopo di questo libro, invece, è quello di affrontare tematiche più complesse, per fornire tutte le informazioni necessarie a creare vere e proprie applicazioni web, piuttosto che un insieme di pagine web. Maggiori informazioni sulle Razor Pages sono disponibili sulla documentazione ufficiale, all'indirizzo http://aspit.co/bo4. Molti dei concetti che introdurremo sono validi anche per le Razor Pages, poiché l'ambiente di esecuzione è il medesimo.

.NET Core e ASP.NET Core riprendono molti dei concetti di .NET Framework e ASP.NET. Se avete già sviluppato applicazioni basate su ASP.NET MVC, per esempio, troverete molti punti di contatto con ASP.NET Core MVC, la versione specificamente pensata per ASP.NET Core.

Perché scegliere ASP.NET Core rispetto ad ASP.NET? I vantaggi sono tantissimi e sono prettamente legati al nuovo modello di sviluppo, già citato. Oltre a un supporto per i più diffusi ambienti di hosting (IIS, nginx, Apache o Docker), uno degli ambiti più apprezzati è sicuramente legato alla modalità di esecuzione del codice. Le versioni del runtime, infatti, possono essere distribuite insieme all'applicazione stessa, senza necessità d'installazione diretta sulla macchina. Questo, soprattutto in scenari di applicazioni basate su container: per esempio, con Docker o sul cloud si traduce in enormi benefici, poiché il container può essere facilmente creato senza dover installare niente sulla macchina host.

Un'altra particolarità di .NET Core, rispetto a .NET Framework, è che tutte le librerie sono distribuite sotto forma di pacchetti NuGet. Questo consente di ottimizzare l'applicazione e di includere le dipendenze necessarie, aggiornandole singolarmente e senza la necessità di aggiornare o distribuire l'intero framework.

> *NuGet è il package manager di riferimento per il mondo .NET. Consente di referenziare facilmente dipendenze attraverso il suo repository centralizzato. Maggiori informazioni sono disponibili su http://www.nuget.org/.*

Un altro vantaggio significativo è che differenti versioni di .NET Core possono essere presenti nello stesso server (o, in generale, nella stessa macchina host), sia installate direttamente sia distribuite insieme all'applicazione stessa. In questo caso, si dice, .NET Core supporta il concetto di installazione side-by-side (fianco a fianco).

Una delle differenze sostanziali tra ASP.NET e ASP.NET Core, che approfondiremo a partire dal Capitolo 4, è la fusione in una sola pipeline di MVC (l'implementazione del pattern Model View Controller) e Web API (il toolkit per creare servizi). Questo rappresenta una semplificazione, poiché diventa più semplice creare endpoint a prescindere dal tipo di ritorno. Come in ASP.NET MVC, anche in ASP.NET Core MVC si utilizza Razor, un linguaggio pensato ad hoc per comporre le View. All'interno di ASP.NET Core, però, Razor ha nuove funzionalità, come i tag helper, che verranno approfondite nei prossimi capitoli.

In estrema sintesi, .NET Core è un ambiente più moderno e, pertanto, dotato di servizi più avanzati rispetto agli analoghi offerti da .NET Framework. Di riflesso, anche ASP.NET Core beneficia di queste migliorie.

ASP.NET Core e .NET Framework

ASP.NET Core può girare sia all'interno di .NET Core sia di .NET Framework. Quest'ultima modalità, però, non consente di creare applicazioni cross-platform, limitando l'ambiente di runtime solo a Windows (caratteristica che viene ereditata da .NET Framework). Questa modalità è in genere utilizzata per poter iniziare a sfruttare alcuni benefici di .NET Core, mantenendo la compatibilità con .NET Framework, per esempio perché alcune librerie non sono state rese compatibili con .NET Core.

In realtà, .NET Core e .NET Framework consentono di sfruttare le stesse librerie, a patto che queste siano state create come librerie .NET Standard. Questo tipo di librerie è stato appositamente pensato per rendere compatibili tra loro librerie pensate per differenti runtime. ASP.NET Core, quindi, di fatto è implementato come una serie di librerie .NET Standard.

A meno che non dobbiate per forza utilizzare una libreria non compatibile, sconsigliamo di utilizzare .NET Framework come runtime e di sfruttare .NET Core, così da poter beneficiare di tutte le funzionalità introdotte da quest'ultimo, come il supporto cross-platform e un runtime più snello e perciò più performante.

Diamo un'occhiata a .NET Standard per cercare di capire meglio come funziona.

.NET Standard

.NET Standard è una specifica che definisce formalmente come le API devono essere implementate all'interno di tutti i runtime .NET ed è stato creato per consentire a differenti runtime di lavorare all'interno di un solo ecosistema.

.NET è basato sullo standard ECMA 335, che però non prevede come debbano comportarsi internamente le librerie della BCL (Base Class Library, l'insieme di librerie di base di .NET).

Per questo motivo, .NET Standard cerca di porre una serie di regole che consentano, a prescindere dall'implementazione del runtime, di avere accesso a un set di API comune. Questo consente agli sviluppatori di creare librerie che siano più facilmente portabili su

runtime differenti, senza dover ricompilare codice o creare differenti versioni compilate di uno stesso output. Le librerie .NET Standard, insomma, sono compatibili in maniera binaria tra tutte le implementazioni che supportano lo standard.

Le diverse versioni di .NET implementano una versione differente di .NET Standard. Ogni versione di .NET Standard porta con sé una serie di API comuni, che se supportata da un determinato runtime, ci garantisce di avere codice che funziona in maniera simile. Al momento di scrivere questo libro, la versione corrente di .NET Standard è la 2.0 e nella tabella 1.1 è riepilogato il supporto delle varie versioni di .NET.

Tabella 1.1 – Versioni di .NET Standard e compatibilità.

.NET Standard	1.6	2.0
.NET Core	1.0	2.0
.NET Framework	4.6.1	4.6.1
Mono	4.6	5.4
Xamarin.iOS	10	10.14
Xamarin.Mac	3.0	3.8
Xamarin Android	7.0	8.0
UWP (Universal Windows Platform)	10.0.16299	10.0.16299

Come si può notare, i runtime che supportano direttamente .NET Standard sono differenti e spaziano da .NET Framework e .NET Core, per arrivare a Mono (una implementazione Open Source di .NET, alla cui base si trovano alcuni prodotti legati a Xamarin), ma anche Xamarin e Universal Windows Platform (UWP).

Creando una libreria basata su .NET Standard 2.0, saremo in grado di farla funzionare su .NET Core 2 o successivo, .NET Framework 4.6.1 o successivo e UWP 10.0.16299 o successivo.

Le versioni di .NET Standard differiscono dalle altre solo per le aggiunte: una volta che una API è confluita in .NET Standard, infatti, non verrà mai più rimossa. Inoltre, le API non vengono modificate.

La gestione di .NET Standard è demandata a un team che decide cosa debba finire all'interno dello standard e cosa no, garantendo una governance dell'intero ecosistema .NET.

Rispetto alle Portable Class Library, che .NET Standard sostituisce, i vantaggi sono tanti. Entrambi consentono di definire un set di API portabili tra runtime differenti, ma .NET Standard ha il vantaggio di non richiedere direttive di compilazione e di essere, come il nome suggerisce, uno standard curato. Come gran parte di quello che ricade intorno a .NET Core, le specifiche sono open source e disponibili su `http://aspit.co/bo5`.

Tecnicamente, una libreria .NET Standard referenzia un metapackage chiamato `NETStandard.Library`, che è in realtà composto da una serie di package NuGet. All'interno, questi pacchetti hanno come target la specifica versione del framework cui fanno riferimento.

> *I metapackage sono una convenzione di NuGet per raggruppare logicamente una serie di package che ha senso stiano insieme. Come vedremo, anche ASP.NET Core è distribuito come metapackage ma resta possibile referenziare direttamente, se necessario, anche i singoli package.*

Creando una libreria di questo tipo, faremo riferimento a una piattaforma ad hoc, chiamata `netstandard`. Ognuna delle versioni è caratterizzata da un nome che ne ricorda la versione. Un esempio di creazione di una libreria di questo tipo è contenuto nel prossimo capitolo. Quando creiamo una libreria per .NET Core, possiamo decidere di crearla sia con .NET Standard, per compatibilità con .NET Framework, sia direttamente per .NET Core, indicando quello che viene chiamato target framework.

Scegliere .NET Core o .NET Framework

A questo punto, una domanda potrebbe essere lecita: quando è meglio scegliere .NET Core e quando .NET Framework? La risposta dipende molto dal tipo di applicazione che andremo a creare. Volendo schematizzare, possiamo dire che .NET Core è ideale quando:

- vogliamo il supporto cross-platform;
- vogliamo sfruttare un toolkit più moderno e in continuo aggiornamento;
- dobbiamo sviluppare un'applicazione con un'architettura a microservice o cloud;
- vogliamo sfruttare i container basati su Docker;
- abbiamo bisogno di performance e scalabilità;
- vogliamo distribuire il runtime insieme all'applicazione stessa.

D'altro canto, .NET Framework è indicato se:

- abbiamo già fatto investimenti su .NET Framework: in questo caso è più indicato non migrare l'applicazione, a meno che uno dei punti precedenti non dovesse risultare determinante;
- ci affidiamo a librerie che non supportano .NET Core;
- abbiamo bisogno di sfruttare componenti nativi di Windows.

In generale, la migrazione di un'applicazione esistente non è sempre semplice, poiché non esiste un percorso (o un tool) preferenziale. Benché gran parte delle API siano compatibili, di fatto, la migrazione vuol dire creare una nuova applicazione da zero e procedere con l'incorporare le funzionalità esistenti, piuttosto che in una migrazione vera e propria, avvicinandosi maggiormente a una riscrittura dell'applicazione stessa.

Pur essendo molto simili tra loro, .NET Core non ha alcuni componenti che in .NET Framework hanno ricoperto un ruolo fondamentale, come gli AppDomain e la Code Access Security, perché fondamentalmente il runtime è differente e le stesse funzionalità di isolamento si possono ottenere isolando i processi o lavorando con i container.

Tra i fattori che possono bloccare il passaggio a .NET Core, vale la pena segnalare la mancanza di alcune funzionalità, che restano appannaggio esclusivo di .NET Framework. Tra queste è d'obbligo segnalare ASP.NET Web Forms, WCF (Windows Presentation Foundation) (in realtà, è presente solo una libreria per creare client, restando impossibile creare la parte server) e WF (Windows Workflow Foundation).

In effetti, in un'applicazione moderna, tutte queste mancanze non si rivelano tali ma è comunque doveroso menzionarle.

Infine, è bene fare una menzione particolare sul linguaggio: non tutti i tipi di progetto sono disponibili per tutti i linguaggi. .NET Core supporta direttamente C#, VB e F# ma questi ultimi sono relegati ad alcuni tipi di applicazioni. Nel caso di VB è solo possibile creare applicazioni console e librerie, mentre F# consente di creare anche applicazioni basate su ASP.NET Core. C# resta il linguaggio con il supporto totale e, anche per questo, oltre che per la sua diffusione, è l'unico linguaggio che tratteremo all'interno di questo libro.

Conclusioni

In questo primo capitolo abbiamo iniziato a inquadrare il funzionamento di ASP.NET Core, dando un'occhiata alle caratteristiche che ci offre .NET Core e ponendo le basi per quello che affronteremo nei prossimi capitoli. Inoltre, abbiamo spiegato le differenze tra .NET Core e.NET Framework, le caratteristiche di .NET Standard e la portabilità di codice tra le varie implementazioni di .NET.

Dobbiamo sempre ricordare che il runtime che sta alla base di tutto, quello che viene chiamato ASP.NET Core, offre caratteristiche molto simili a quanto offerto da ASP.NET, e che uno sviluppatore che conosca bene ASP.NET ha il vantaggio di poter partire conoscendo bene gran parte dei prerequisiti. A proposito di prerequisiti, è opportuno sottolineare ancora una volta che è assolutamente necessario conoscere C# e le relative tecnologie, come, per esempio, LINQ, e i rudimenti legati al web, come il protocollo HTTP e le specifiche legate a HTML, CSS e JavaScript.

Al momento di scrivere, la versione rilasciata è la 2.1, ma è stata già annunciata la 2.2, attesa per la fine del 2018. Non si tratta di una versione che aggiunge moltissime funzionalità, quindi la maggior parte degli argomenti trattati sono validi anche per questa nuova release.

Ora che abbiamo gettato le basi, possiamo partire con i primi esperimenti. Nel prossimo capitolo inizieremo ad affrontare le caratteristiche di .NET Core, del relativo SDK e dei tool di sviluppo, così da iniziare a trattare le funzionalità di base e poter affrontare più facilmente gli argomenti più avanzati contenuti nel resto di questo libro.

2
Usare .NET Core CLI, Visual Studio e Visual Studio Code per iniziare con ASP.NET

Ora che abbiamo l'idea di quali siano gli ambiti di utilizzo di .NET Core, possiamo entrare nel merito degli strumenti che ci permettono di costruire le prime semplici applicazioni web che si baseranno su ASP.NET Core.

In passato, iniziare a lavorare con ASP.NET significava dotarsi di un PC Windows e installare una versione aggiornata di .NET Framework. Inoltre, lo sviluppatore doveva dotarsi di Visual Studio, l'IDE (Integrated Development Environment) prodotto da Microsoft. Il tempo richiesto per preparare il PC di sviluppo poteva misurarsi in ore.

A partire da .NET Core 1.0, rilasciato nel giugno 2016, Microsoft presenta un framework più snello e modulare per costruire applicazioni web (e non solo). Questo nuovo framework non è più legato alla sola piattaforma Windows e non richiede necessariamente l'uso di un ambiente di sviluppo avanzato come Visual Studio. Scrivere la prima riga di codice è un'attività che si può completare in minuti, a tutto vantaggio sia di coloro che si avvicinano per la prima volta alla programmazione sia degli sviluppatori più esperti.

.NET Core è un prodotto per tutti, multipiattaforma e open-source, che incontra le esigenze e le abitudini di una porzione più ampia della community di sviluppatori.

Scegliere l'ambiente di sviluppo

Dal momento che Microsoft si è aperta a un pubblico più vasto ed eterogeneo, è importante che ciascuno sviluppatore possa contare su un'esperienza di sviluppo soddisfacente, a prescindere da quale sia la propria piattaforma preferita.

Gli sviluppatori che sono abituati a lavorare su Windows possono continuare a usare una IDE completa come Visual Studio, disponibile anche nell'edizione Community, scaricabile gratuitamente da http://www.visualstudio.com.

Per loro, così come per gli sviluppatori che prediligono Linux e Mac, si apre un'ulteriore possibilità: usare un editor leggero, multipiattaforma e facile da apprendere come Visual Studio Code, che offre ugualmente molti ausili essenziali alla scrittura del codice, pur essendo privo di alcune funzionalità avanzate, tra cui gli strumenti di diagnostica delle performance.

Per gli utenti Mac, è inoltre disponibile Visual Studio for Mac, che mira a offrire le stesse funzionalità di Visual Studio e viene proposto con lo stesso modello di licensing, consultabile all'indirizzo: http://aspit.co/bkj.

Tutti gli ambienti di sviluppo menzionati sono in grado di garantire un'ottima produttività, così che lo sviluppatore sia libero di usare quello che preferisce. Qualsiasi sia la scelta, gli sarà possibile creare applicazioni ASP.NET Core di qualità, anche nel lavoro in team: ogni edizione di Visual Studio include infatti il supporto nativo a Git, di gran lunga la tecnologia più diffusa per il controllo di versione.

IDE o editor?

Per chi si avvicina la prima volta alla programmazione, si pone il problema di capire quale sia lo strumento più adatto. Le scelte non riguardano soltanto la piattaforma ma anche se sia più opportuno iniziare con una IDE di sviluppo avanzata (Visual Studio o Visual Studio for Mac) o se invece iniziare a muovere i primi passi con un editor semplice, di utilizzo più immediato (Visual Studio Code). La Tabella 2.1 prova a fornire un aiuto a chi è in cerca di consiglio.

Tabella 2.1 – Guida alla scelta dell'ambiente di sviluppo.

Scenario	Ambiente consigliato
Studente o sviluppatore alla prima esperienza	Visual Studio Code
Sviluppatore con esperienza in altre tecnologie web (es. PHP, Node.js, Ruby) che intende valutare .NET Core per la prima volta	Visual Studio Code
Sviluppatore con esperienza di .NET Framework o che deve manutenere applicazioni realizzate con .NET Framework	Visual Studio
Sviluppatore con esperienza, già abituato all'uso di IDE come IntelliJ Idea, Eclipse o NetBeans	Visual Studio o Visual Studio for Mac
Sviluppatore che intende creare applicazioni mobile con Xamarin, oltre ad applicazioni ASP.NET Core	Visual Studio o Visual Studio for Mac

Ora che abbiamo iniziato a capire come dotarci di una IDE o di un buon editor, possiamo continuare a esplorare le caratteristiche di .NET Core, affrontando il tema relativo all'SDK.

Installare il .NET Core SDK

Per iniziare a sviluppare applicazioni per .NET Core, dobbiamo visitare l'indirizzo http://aspit.co/bkc e effettuare il download del pacchetto di installazione.

La pagina permette i download per tutte le piattaforme supportate (Windows, macOS e varie distribuzioni Linux). Scorrendo la pagina, troviamo varie altre opzioni, come viene illustrato nella Figura 2.1.

È sempre importante saper scegliere l'installer più adatto alle nostre esigenze. Una prima distinzione consiste nel tipo di rilascio:

- Long Term Support (LTS): consiste di rilasci stabili che godono di un supporto di tre anni da parte di Microsoft. Sono i rilasci più indicati da usare in applicazioni mission-critical, o dovunque sia preferibile costruire la propria applicazione su un framework consolidato e durevole, che riceve comunque aggiornamenti di sicurezza quando necessario. Tipicamente, una nuova major release LTS viene resa disponibile dopo almeno un anno dalla precedente.
- Current: consiste dei rilasci più recenti e stabili di .NET Core. Sono consigliati per gli sviluppatori che vogliano valutare le funzionalità più recenti e fornire feedback affinché siano ancora più raffinate prima del loro ingresso in una successiva versione LTS. I rilasci Current si susseguono con un ritmo molto frequente (di solito ogni 1-3 mesi), in cui ricevono aggiornamenti sia alle funzionalità sia alla sicurezza. Ogni versione Current continua a essere supportata da Microsoft per tre mesi dopo un successivo rilascio Current e per un anno da un successivo rilascio LTS.

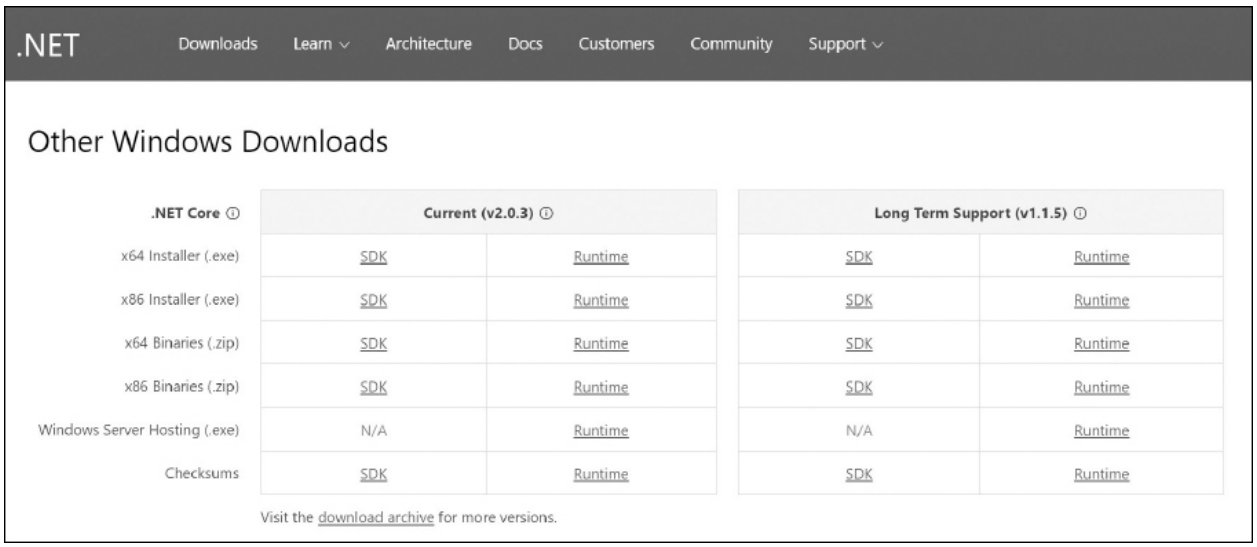


Figura 2.1 – Opzioni di download di .NET Core.

Inoltre, il .NET Core viene distribuito in due edizioni:

- Runtime: è indicata per gli ambienti di produzione, ovvero per macchine server che ospitano le applicazioni per il pubblico di utenti. Contiene .NET Core propriamente detto.
- SDK: è indicata per i PC e Mac di sviluppo. Può contenere una o più versioni di .NET Core e, in aggiunta, fornisce il .NET Core CLI, ovvero gli strumenti da riga di comando necessari a creare, gestire, compilare ed eseguire progetti.

Il .NET Core SDK è la versione da installare quando vogliamo iniziare lo sviluppo di applicazioni. Sta a noi la scelta se procurarci una versione LTS o Release.

L'installer copierà i file nei seguenti percorsi, come indicato nella Tabella 2.2.

Tabella 2.2 – Percorsi d'installazione del .NET Core SDK.

Sistema operativo	Percorso di installazione
Windows	C:\Program Files\dotnet
Linux	A seconda della modalità scelta /usr/bin/dotnet oppure ~/.dotnet
macOS	/usr/local/share/dotnet

In qualsiasi momento possiamo installare nuove versioni del .NET Core SDK che coesisteranno fianco a fianco. Ogni versione è isolata dall'altra in quanto risiede in una specifica sottodirectory nel percorso di installazione. Questo ci permette di provare a usare nuove versioni senza rischi e di scegliere di volta in volta la versione più adatta al progetto che andremo a intraprendere.

Gestire progetti ASP.NET Core con il .NET Core CLI

Prima di iniziare a creare la nostra prima applicazione web, dobbiamo acquisire familiarità con .NET Core SDK, che consente di creare, compilare ed eseguire i progetti realizzati per .NET Core. Per essere il più inclusiva possibile, Microsoft fornisce il .NET Core CLI (Command-Line Interface), ovvero uno strumento da riga di comando disponibile su ogni sistema operativo supportato dal .NET Core SDK. Imparare a usare la riga di comando è importante anche per coloro che si avvalgono regolarmente di una IDE di sviluppo, che con apposite interfacce grafiche permette di eseguire le stesse operazioni in maniera visuale. Infatti, dato che la piattaforma di sviluppo e quella di produzione possono essere diverse, così come le piattaforme e gli strumenti di sviluppo usati dai nostri collaboratori, è importante conoscere i comandi del .NET Core CLI per essere produttivi anche laddove Visual Studio non sia disponibile.

Guida all'utilizzo del comando dotnet

Il comando `dotnet` è il nostro punto di ingresso a tutte le funzionalità di gestione del progetto, richiamabili con vari sottocomandi (o "verbi"). Per impiegare al meglio questo strumento, possiamo contare su una guida in linea, sensibile al contesto, che viene visualizzata aggiungendo l'opzione `--help` (o `-h` per brevità) al comando che stiamo componendo. Digitiamo quanto segue da riga di comando per avere una panoramica delle possibilità offerte.

Esempio 2.1

```
dotnet --help
```

In output appaiono i sottocomandi esistenti, come nella Tabella 2.3. Li affronteremo nel corso di questo capitolo.

Tabella 2.3 – I sottocomandi di dotnet per la gestione della soluzione.

Sottocomando	Descrizione
`new`	Inizializza i progetti .NET
`restore`	Ripristina le dipendenze specificate nel progetto .NET
`run`	Compila ed esegue immediatamente un progetto .NET
`build`	Compila un progetto .NET
`publish`	Pubblica un progetto .NET per la distribuzione self-contained (includendo il runtime)
`test`	Esegue gli unit test con l'istanza di Test Runner specificata nel progetto
`pack`	Crea un pacchetto NuGet
`migrate`	Esegue la migrazione di un progetto basato su project.json in uno basato su MSBuild
`clean`	Pulisce gli output di compilazione
`sln`	Consente di modificare i file solution (SLN)
`add`	Consente di aggiungere un riferimento a un progetto o a un pacchetto NuGet
`remove`	Consente di rimuovere un riferimento a un progetto o a un pacchetto NuGet
`list`	Elenca i riferimenti del progetto corrente ad altri progetti
`nuget`	Comandi aggiuntivi per pubblicare o rimuovere pacchetti da un server NuGet
`msbuild`	Esegue Microsoft Build Engine (MSBuild)
`vstest`	Esegue lo strumento da riga di comando per l'esecuzione di test Microsoft

Oltre a permettere la gestione dei progetti in sé, il comando `dotnet` offre ulteriori sottocomandi – elencati nella Tabella 2.4 – che ci aiutano ad automatizzare determinati compiti. Imparare a usare questi ausili può ridurre o eliminare attività ripetitive, come riavviare manualmente l'applicazione in seguito a una modifica al codice sorgente.

Tabella 2.4 – I sottocomandi di dotnet per la strumentazione e gli ausili allo sviluppo.

Sottocomando	Descrizione

`store`	Inserisce gli assembly specificati nell'archivio pacchetti di runtime
`tool`	Installa tool di sviluppo a livello globale, in maniera simile ai tool npm
`buildserver`	Interagisce con i server long-running che restano attivi dopo una compilazione
`dev-certs`	Crea e installa un certificato SSL auto-firmato da usare durante lo sviluppo
`ef`	Espone operazioni inerenti a Entity Framework Core, come scaffolding e migration
`sql-cache`	Crea una tabella e i relativi indici su SQL Server a uso di cache distribuita
`watch`	Riavvia automaticamente l'applicazione quando un file sorgente cambia
`help`	Apre la documentazione online su docs.microsoft.com per uno specifico comando

Usare la guida è sempre un modo efficace e immediato per apprendere l'utilizzo del comando `dotnet`. Il comando fallisce quando lo inviamo senza aver fornito alcuni dei suoi argomenti obbligatori, oppure quando li abbiamo forniti in maniera non corretta. In questa situazione, la guida viene mostrata automaticamente, anche senza che sia indicata l'opzione `--help`, come si vede nella Figura 2.2.

```
C:\>dotnet sln add
You must specify at least one project to add.
Usage: dotnet sln <SLN_FILE> add [options] <args>

Arguments:
  <SLN_FILE>   Solution file to operate on. If not specified, the command will search the current directory for one.
  <args>       Add one or more specified projects to the solution.

Options:
  -h, --help   Show help information.
```

Figura 2.2 – Output di un comando fallito su .NET Core CLI sotto Windows.

Nell'output si può osservare quanto segue:

- Un messaggio in rosso che indica la causa del fallimento del comando.
- Un esempio di corretto utilizzo del comando. Gli argomenti obbligatori sono tipicamente rappresentati tra parentesi angolari, mentre le opzioni tra parentesi quadre. Il comando presentato nella figura è appunto fallito per la mancanza di un argomento obbligatorio. Le parentesi hanno il solo scopo di indicare l'obbligatorietà (o la non obbligatorietà) dell'argomento e non vanno digitate al successivo invio del comando.
- Sono riepilogati gli argomenti e le opzioni disponibili, con relativa spiegazione.

Selezionare una versione del .NET Core SDK

A poco più di un anno dal primo rilascio ufficiale di .NET Core, Microsoft ha già presentato .NET Core 2.1, che introduce importanti miglioramenti riguardanti prestazioni, stabilità e funzionalità.

Col passare del tempo, sarà normale voler installare sul nostro PC o Mac di sviluppo varie versioni del .NET Core SDK. Per ciascun progetto manteniamo la libertà di impiegare una specifica versione tra quelle installate. Vediamo come.

Elencare le SDK installate e determinare quella in uso

Dalla versione 2.1 del .NET Core SDK, il comando `dotnet` si arricchisce di una nuova opzione `--list-sdks`, che usiamo per elencare tutte le versioni installate nel sistema.

Esempio 2.2

```
dotnet --list-sdks
```

Di tutte le versioni elencate, soltanto la più recente risulterà essere quella "attiva". Per verificarlo, usiamo l'opzione `--version`.

Esempio 2.3

```
dotnet --version
```

Per ottenere ancora più informazioni sulla piattaforma corrente, usiamo invece l'opzione `--info`.

Esempio 2.4

```
dotnet --info
```

La Figura 2.3 mostra l'output per la versione 2.1.0 del .NET Core SDK installata su un PC Windows 10.

```
Prompt dei comandi
C:\>dotnet --info
.NET Core SDK (che rispecchia un qualsiasi file global.json):
 Version:   2.1.300
 Commit:    adab45bf0c
```

Figura 2.3 – Esempio di output del comando `dotnet --info`.

Cambiare la versione del .NET Core SDK con il global.json

Per usare una versione del .NET Core SDK diversa da quella attiva, possiamo facilmente reimpostarla per il progetto corrente creando un file global.json nella sua directory. Il file può anche essere creato con il comando `dotnet new globaljson` e il suo contenuto è illustrato dall'Esempio 2.5.

Esempio 2.5

```
{
   "sdk": {
   "version": "2.1.0"
   }
}
```

In questo caso abbiamo indicato la 2.1.0 ma, se questa specifica versione non fosse installata nel nostro sistema, verrebbe comunque scelta un'altra SDK con una minor-version uguale o successiva, come per esempio la 2.1.1 o la 2.2.0. In nessun caso verrebbe scelta una major successiva, come un'eventuale 3.0.0, dato che potrebbe presentare importanti breaking change. Dopo aver creato il file, verifichiamo quale sia l'effettiva versione selezionata con il comando `dotnet --version` mentre siamo posizionati nella directory del progetto. Ogni successivo comando verrà eseguito usando l'SDK indicata.

Questa funzionalità, denominata minor-version roll-forward, ha effetto anche in fase di deploy: se l'applicazione è stata compilata usando .NET Core 2.0.0, riuscirà a funzionare senza modifiche anche su macchine server in cui è stato installato .NET Core 2.1.0, ovvero una versione minor successiva.

Il primo progetto con ASP.NET Core

Ora che abbiamo una comprensione base di come muoverci con il .NET Core CLI, possiamo usare il comando `dotnet` per creare il nostro primo progetto e per compiere su di esso quelle operazioni tipiche che lo accompagneranno durante tutto il suo ciclo di vita:

- la creazione del progetto a partire da un template di esempio;
- la compilazione del codice sorgente del progetto, un passo necessario per poter eseguire l'applicazione;
- l'esecuzione dell'applicazione, che ci consentirà di verificarne il funzionamento;
- l'aggiunta di pacchetti NuGet, per poter estendere le funzionalità del nostro progetto con librerie sviluppate da Microsoft e da terze parti;
- la creazione di altri progetti correlati, per strutturare al meglio la nostra applicazione;
- la pubblicazione del progetto, affinché sia preparato per essere eseguito su macchine server con sistemi operativi e/o architetture diverse da quella del nostro PC o Mac di sviluppo.

Scegliere un template di progetto

Per iniziare a sviluppare una nuova applicazione, è sempre preferibile partire da un template, ovvero da una bozza di applicazione minimale (ma funzionante) che costituisce il fondamento per ciò che andremo a costruire. Il .NET Core SDK mette a disposizione vari template, per vari tipi di applicazioni diverse. Lanciamo il seguente comando:

Esempio 2.6

```
dotnet new
```

Così digitato, il comando `dotnet new` è incompleto poiché mancante del nome del template. La guida in linea ci supporta elencando in output tutti i template disponibili. La Tabella 2.5 riepiloga i template d'interesse per le applicazioni ASP.NET Core.

Tabella 2.5 – I template d'interesse per le applicazioni ASP.NET Core.

Nome template	Linguaggi	Descrizione
`web`	C#, F#	Progetto ASP.NET Core vuoto
`mvc`	C#, F#	Progetto ASP.NET Core MVC
`razor`	C#	Progetto ASP.NET Core con Razor Pages

`webapi`	C#, F#	Progetto ASP.NET Core Web API
`angular`	C#	Progetto ASP.NET Core con Angular
`react`	C#	Progetto ASP.NET Core con React.js
`reactredux`	C#	Progetto ASP.NET Core con React.js e Redux
`classlib`	C#, F#, VB	Class Library
`mstest, xunit`	C#, F#, VB	Progetto di Unit Test
`sln`		File di Solution
`globaljson`		File global.json usato per indicare la versione dell'SDK
`nugetconfig`		File nuget.config per la configurazione dei feed NuGet
`webconfig`		File web.config per l'integrazione e la configurazione di IIS

Possiamo notare che non tutti i template sono ancora disponibili per tutti i linguaggi. Il supporto a VB.NET, nella versione 2.1 del .NET Core SDK, risulta essere ancora acerbo e limitato alle applicazioni console. È certamente possibile realizzare anche applicazioni ASP.NET Core con questo linguaggio, anche se è preferibile attendere che la strumentazione offra un'esperienza di utilizzo equiparabile a quella di C#.

Per scegliere un template tra i tanti disponibili, dovremmo considerare il tipo di applicazione che intendiamo sviluppare.

- Per siti e applicazioni web realizzati secondo il pattern Model View Controller (presentato nel Capitolo 4) scegliamo il template `mvc`. Per lo stesso tipo di applicazione – possiamo anche scegliere di avvalerci delle Razor Pages – usiamo il template `razor`.
- Per applicazioni che espongono unicamente servizi REST e agiscono da backend per altre applicazioni client (che siano desktop, web o mobile) scegliamo il template `webapi`.
- Per Single Page Applications, che fanno un ampio uso di JavaScript nel realizzare un'esperienza d'uso desktop-like, scegliamo i template `angular`, `react` o `reactredux`, in base al framework che intendiamo utilizzare per lo sviluppo della parte client dell'applicazione web.
- Per un'applicazione ASP.NET Core minimale, che non si avvale di alcuno dei pattern menzionati in precedenza, usiamo il template `web`. Grazie a questo template, privo di servizi accessori, possiamo avere un'idea di quali siano le parti essenziali di un'applicazione ASP.NET Core, che esamineremo più approfonditamente nel Capitolo 3.

Supponendo di aver scelto di iniziare il nostro progetto usando il template `mvc` e il linguaggio C# (il predefinito), digitiamo il seguente comando dopo esserci posizionati in una directory vuota:

Esempio 2.7

```
dotnet new mvc
```

Nel caso in cui volessimo creare il progetto con un linguaggio differente o in una sottodirectory diversa da quella corrente, lanciamo lo stesso comando fornendo gli argomenti `--language` (o `-lang` per brevità) e `--output` (o `-o` per brevità).

Esempio 2.8

```
dotnet new mvc --language F# --output MyProject
```

Non dimentichiamo che possiamo usare l'opzione `--help` (o `-h` per brevità) in coda a qualsiasi comando per visualizzare la guida in linea. In questo caso è particolarmente importante farlo, perché i template di progetto dispongono essi stessi di opzioni di configurazione.

Esempio 2.9

```
dotnet new mvc --help
```

Grazie alla guida in linea, riusciamo a scoprire che il template `mvc` dispone di alcune opzioni proprie, tra cui quelle per indicare la modalità di autenticazione che desideriamo supportare per i nostri utenti. Supponendo di voler creare un'applicazione che supporti l'accesso con username e password da parte di utenti persistiti in un database locale, digitiamo il comando riportato nell'esempio che segue.

Esempio 2.10

```
dotnet new mvc --auth Individual
```

Le varie modalità di autenticazione saranno trattate più approfonditamente nel Capitolo 17. Nel frattempo, esaminiamo l'output del comando, che appare come nella Figura 2.4.

```
C:\MyFirstWebApp>dotnet new mvc --auth Individual
Getting ready...
The template "ASP.NET Core Web App (Model-View-Controller)" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/template-3pn for details.

Processing post-creation actions...
Running 'dotnet restore' on C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restoring packages for C:\MyFirstWebApp\MyFirstWebApp.csproj...
  Restore completed in 84,59 ms for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Restore completed in 1,48 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Restore completed in 1,55 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.
  Generating MSBuild file C:\MyFirstWebApp\obj\MyFirstWebApp.csproj.nuget.g.props.
  Generating MSBuild file C:\MyFirstWebApp\obj\MyFirstWebApp.csproj.nuget.g.targets.
  Restore completed in 2,88 sec for C:\MyFirstWebApp\MyFirstWebApp.csproj.
```

Figura 2.4 – Output del comando di creazione di un progetto ASP.NET MVC con account locali.

Alla creazione del progetto, vari file verranno posizionati all'interno della directory corrente. Esamineremo nel dettaglio i contenuti della directory nel Capitolo 3.

Dalla versione 2.0 del .NET Core SDK, il comando `dotnet new` ottiene automaticamente gli eventuali pacchetti NuGet referenziati dal progetto, in modo che sia subito possibile avviare l'applicazione, senza dover eseguire altri passi intermedi.

Creare un progetto dagli ambienti di sviluppo

Il .NET Core CLI è uno strumento largamente impiegato quando si lavora con un editor di codice come Visual Studio Code, che espone la riga di comando da un apposito riquadro, come si può vedere nella Figura 2.5.

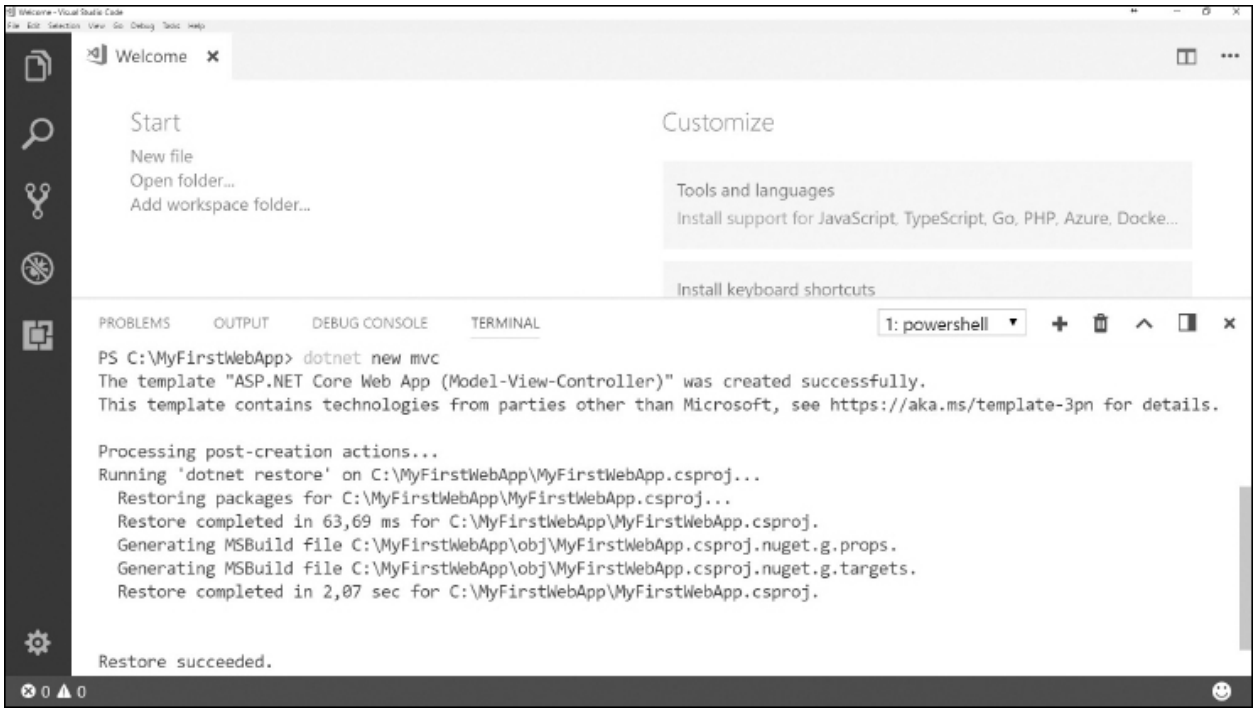


Figura 2.5 – Creazione di un progetto dal terminale integrato in Visual Studio Code.

Se preferiamo un approccio visuale, Visual Studio ci permette di accedere al wizard di creazione del progetto dal menu `File > New > Project`. Da lì, selezioniamo `ASP.NET Core Web Application` e clicchiamo l'icona di uno dei template disponibili, come si può notare nella Figura 2.6.

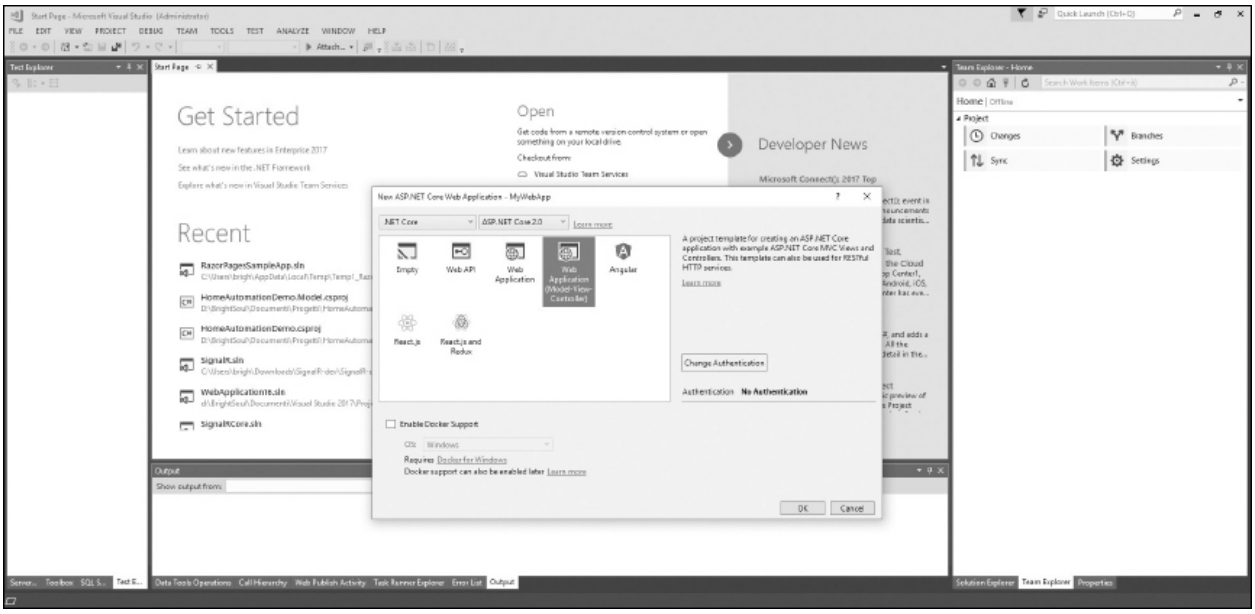


Figura 2.6 – Scelta di un template di progetto per applicazioni ASP.NET Core da Visual Studio.

Anche Visual Studio for Mac offre una creazione di progetto guidata, come si può notare nella Figura 2.7.

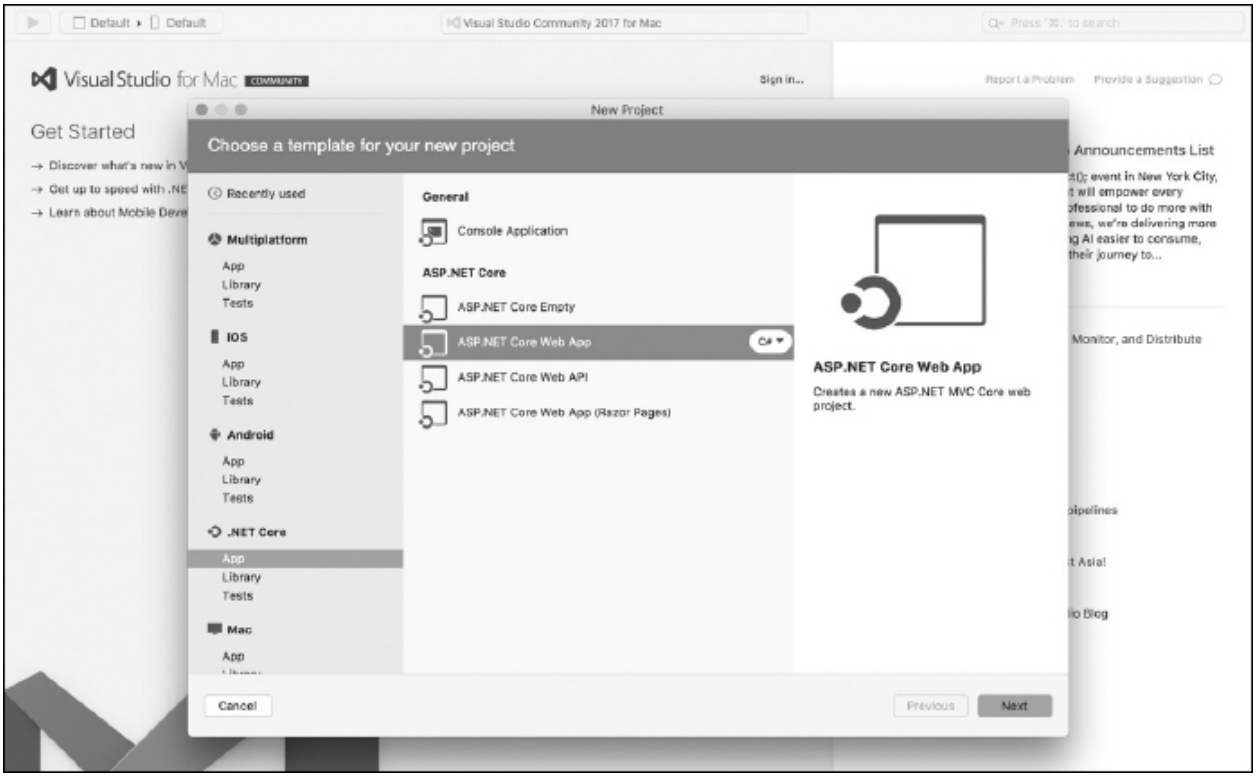


Figura 2.7 – Scelta di un template di progetto per applicazioni ASP.NET Core da Visual Studio for Mac.

Nel caso in cui i template forniti con .NET Core non fossero perfettamente idonei al nostro scenario, possiamo procurarcene altri installandoli da riga di comando.

Installare altri template di progetto

Moltissimi altri template possono essere aggiunti a quelli già elencati. Microsoft ha messo a disposizione un apposito sito da cui scaricare altri template, molti dei quali sviluppati da terze parti, spesso a scopo didattico. Il sito è raggiungibile all'indirizzo: `http://aspit.co/bkd`.

Come esempio, installiamo un template per applicazioni ASP.NET Core Web Api, che include una documentazione auto-generata con Swagger, un progetto di unit test e una configurazione per Docker.

Esempio 2.11

```
dotnet new -i "Popov1024.HttpApi.Template.CSharp::*"
```

Ora, il nostro elenco di template risulterà aggiornato e potremo creare il nuovo progetto digitando:

Esempio 2.12

```
dotnet new httpapi
```

I template sono un ottimo strumento di auto-apprendimento perché ci illustrano il modo corretto di usare un framework o una tecnologia. Sia Microsoft sia autori di prodotti di terze parti mettono a disposizione template per guidarci con l'esempio e consentici di iniziare a usare le tecnologie nel modo più rapido possibile.

Compilare ed eseguire l'applicazione

Dopo aver creato il progetto, possiamo sin da subito compilare il suo codice sorgente, in modo da produrre gli assembly (i file con estensione .dll) che contengono il codice binario eseguibile sulla nostra macchina di sviluppo. A tale scopo, lanciamo il comando `dotnet build`, eventualmente fornendo una configurazione al compilatore mediante il parametro `--configuration` (o `-c` per brevità). Per default, abbiamo già a disposizione due configurazioni tipiche:

- Debug: è la configurazione predefinita, ideale da usare durante lo sviluppo dell'applicazione. Prevede l'emissione di simboli di debug (file con estensione .pdb), che mantengono una correlazione tra codice sorgente e codice compilato, affinché sia semplice per lo sviluppatore risalire alla riga di codice sorgente che sta causando un problema.
- Release: è la configurazione consigliata per compilare un'applicazione da mettere in produzione. Il compilatore ha la libertà di effettuare delle ottimizzazioni per migliorare le performance dell'applicazione, pur mantenendo inalterata la sua logica di funzionamento. I simboli di debug non sono emessi.

Segue un esempio del comando di compilazione in modalità Release.

Esempio 2.13

```
dotnet build --configuration Release
```

La compilazione dell'applicazione può avere successo solamente se non abbiamo commesso errori sintattici nella digitazione nel codice. Nella Figura 2.8, osserviamo l'output del comando `dotnet build` che segnala un errore alla riga 36 del file di codice `Startup.cs`, che dovremmo correggere prima di ritentare la compilazione.

```
C:\MyFirstWebApp>dotnet build
Microsoft (R) Build Engine version 15.4.8.50001 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Startup.cs(36,41): error CS1010: Newline in constant [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1026: ) expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1002: ; expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]

Build FAILED.

Startup.cs(36,41): error CS1010: Newline in constant [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1026: ) expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
Startup.cs(36,55): error CS1002: ; expected [C:\MyFirstWebApp\MyFirstWebApp.csproj]
    0 Warning(s)
    3 Error(s)

Time Elapsed 00:00:06.75
```

Figura 2.8 – Segnalazione di un errore di compilazione.

La corretta compilazione è un passaggio obbligatorio affinché l'applicazione possa essere avviata. Il comando `dotnet run`, che si occupa di avviare l'applicazione, eseguirà automaticamente la compilazione nel caso in cui non fosse stata già eseguita esplicitamente. Anche in questo caso, ci è possibile fornire il nome della configurazione.

Esempio 2.14

```
dotnet run -c Release
```

In fase di avvio, l'applicazione ASP.NET Core avvia il webserver Kestrel (verrà trattato nel Capitolo 22) che, per default, si pone in ascolto sulla porta TCP 5000. Come vedremo nel prossimo capitolo, la porta TCP e le interfacce di rete usate dal server possono essere facilmente configurate. La Figura 2.9 mostra l'output prodotto all'avvio dell'applicazione.

```
C:\MyFirstWebApp>dotnet run
Hosting environment: Production
Content root path: C:\MyFirstWebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figura 2.9 – Il web server Kestrel si mette in ascolto di connessioni all'avvio dell'applicazione ASP.NET Core.

L'output del comando `dotnet run` ci offre una chiara indicazione che l'applicazione web ASP.NET Core è ora pronta a ricevere richieste HTTP. Apriamo il nostro browser preferito e digitiamo l'indirizzo `http://localhost:5000/` per veder apparire la nostra prima applicazione ASP.NET Core. Ciò che si vede nella Figura 2.10 è opera del template di progetto `mvc` che abbiamo indicato al comando `dotnet new`.

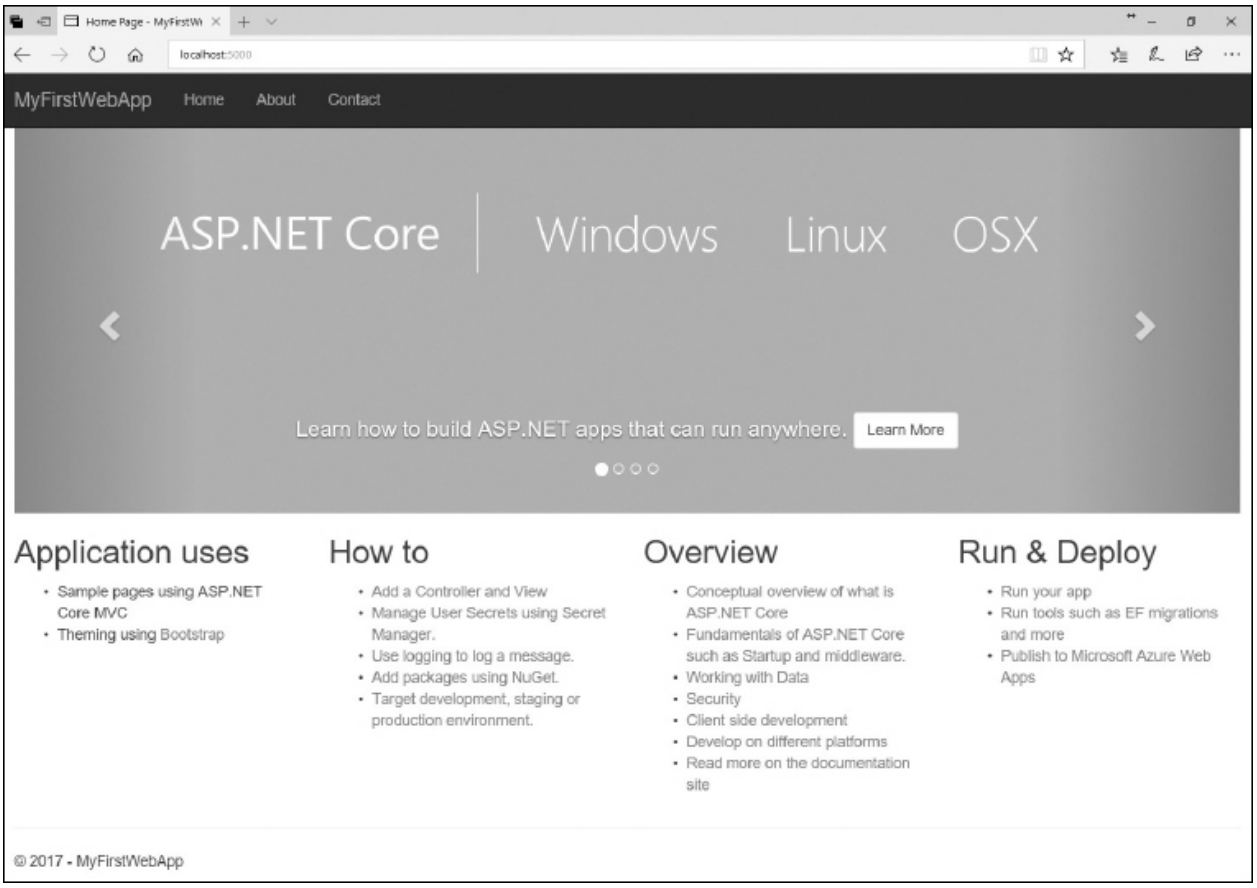


Figura 2.10 – Aspetto dell'applicazione ASP.NET Core creata dal template mvc.

Le applicazioni ASP.NET Core sono applicazioni console

L'aver avviato l'applicazione da riga di comando ci dà un indizio inequivocabile: stiamo avendo a che fare con un'applicazione console. Questa caratteristica la rende indipendente da altri servizi della piattaforma corrente, e perciò autonoma nel funzionamento. Si dice quindi che l'applicazione è self-hosted, cioè possiede tutto lo stack tecnologico di cui ha bisogno, come avremo modo di verificare nel Capitolo 3, esaminando il suo entry point e le opzioni di configurazione.

Proprio come le altre applicazioni console, un'applicazione ASP.NET Core può accettare argomenti da riga di comando, a nostra discrezione. Per evitare che gli argomenti dell'applicazione possano essere confusi con quelli propri del comando `dotnet run`, dobbiamo aver cura di porli in coda, usando la sintassi `chiave=valore`. Nell'esempio che segue, avviamo l'applicazione fornendo l'argomento `-c`, interpretato dal comando `dotnet`, e un nostro argomento personalizzato `log` che verrà invece inoltrato dall'applicazione.

Esempio 2.15

```
dotnet run -c Release log=info
```

È anche possibile aggiungere il supporto ad argomenti che usino i prefissi `-`, `--` o `/`. Per l'approfondimento, si rimanda al paragrafo switch mappings della documentazione ufficiale, raggiungibile all'indirizzo: `http://aspit.co/bke`.

Debug dell'applicazione

Eseguire l'applicazione ci permette indubbiamente di provarne il funzionamento ma, durante lo sviluppo, abbiamo bisogno di uno strumento più preciso, che ci permetta di ispezionare il codice nel momento stesso in cui viene eseguito, eventualmente bloccandone l'esecuzione in determinati punti per verificare che le condizioni in cui si trova a operare siano quelle attese.

Quello di cui abbiamo bisogno è un debugger, ovvero un componente usato dagli editor di codice per osservare le applicazioni durante la loro esecuzione e conferire allo sviluppatore preziose informazioni di diagnostica. Vediamo come sfruttare il debugger dal nostro editor preferito.

Debug con Visual Studio

Dopo aver aperto un progetto dal menu `File > Open > Project/Solution`, rivolgiamo la nostra attenzione alla toolbar situata nella parte alta di Visual Studio. In essa si trova un bottone che reca un'icona "play" in verde e il testo "IIS Express". Tale bottone, anche attivabile da tastiera con il tasto F5, permette l'avvio dell'applicazione in modalità debug.

IIS Express è il web server di sviluppo fornito con l'installazione di Visual Studio e offre le stesse funzionalità di IIS, che tipicamente si troviamo nelle macchine server che ospiteranno l'applicazione. In questo modo, possiamo provare l'applicazione in un ambiente quanto più simile a quello di produzione sin dalle prime fasi di sviluppo.

IIS Express non sostituisce in alcun modo il web server Kestrel di cui abbiamo parlato in precedenza. Piuttosto, si pone difronte a esso per offrire ulteriori funzionalità, come caching e url rewriting, per poi inoltrare le richieste a Kestrel, affinché siano gestite dall'applicazione ASP.NET Core.

La Figura 2.11 mostra il bottone per l'avvio del debug, che ci permette anche di selezionare il browser in cui l'applicazione verrà visualizzata.

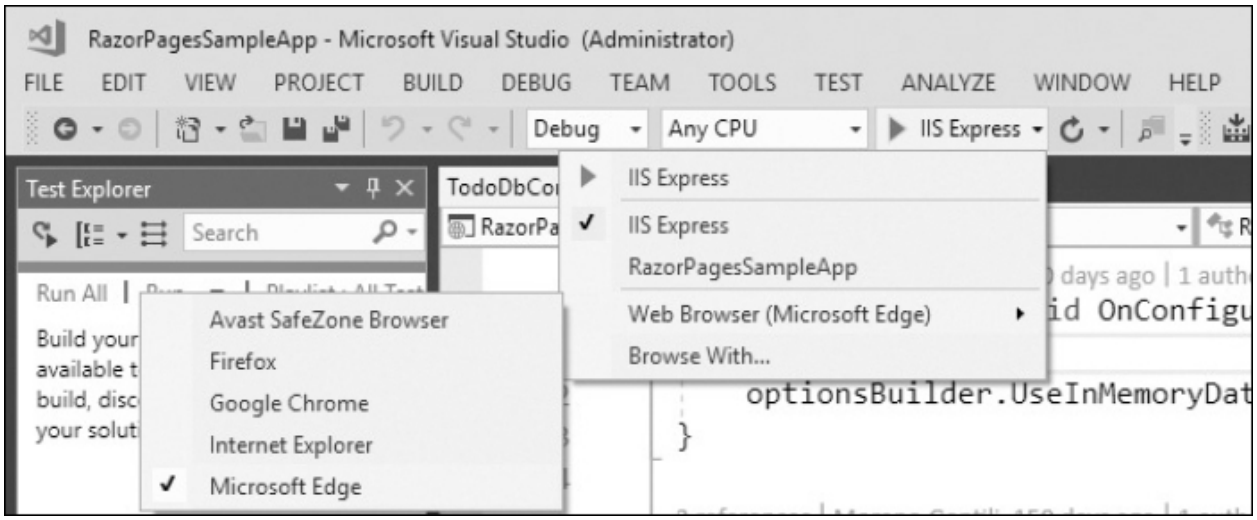


Figura 2.11 – Bottone per l'avvio in debug di un'applicazione ASP.NET Core da Visual Studio.

L'editor di codice dispone di una banda grigia lungo il lato sinistro, che possiamo cliccare per impostare un breakpoint, ovvero un punto in cui l'esecuzione dell'applicazione si metterà in pausa per darci modo di ispezionare oggetti e variabili locali, semplicemente portando il mouse su di essi. I breakpoint sono rappresentati da un cerchio rosso situato in corrispondenza della riga di codice, come viene illustrato

```
25    async Task ITodoRepository.Add([Bind(nameof(Todo.Description))] Todo todo)
26    {
27        Todos.Add(todo);
28        await SaveChangesAsync();
29    }
30
```

nella Figura 2.12.

Figura 2.12 – Impostazione di un breakpoint in corrispondenza di una riga di codice.

Mentre l'esecuzione è in pausa, è anche possibile apportare modifiche al codice. Grazie a questa funzionalità, denominata Edit and Continue, Visual Studio ci rende più produttivi e riduce lo spreco di tempo derivante dal dover ricompilare l'intera applicazione a seguito di piccole modifiche. La Figura 2.13 dimostra come sia possibile intervenire sul codice durante il debug.

```
25    async Task ITodoRepository.Add([Bind(nameof(Todo.Description))] Todo todo)
26    {
27        if (string.IsNullOrEmpty(todo.Description))
28            throw new
29        Todos.Add(todo);
30        await SaveChangesAsync();
```

Figura 2.13 – Con la funzionalità Edit and Continue possiamo modificare il codice durante l'esecuzione.

Quando siamo pronti a riprendere l'esecuzione, premiamo di nuovo il bottone "Continue" o il tasto F5, per proseguire fino al prossimo breakpoint (se presente). In alternativa, possiamo avanzare passopasso (un'istruzione alla volta) con il tasto F10 oppure con il tasto F11, se desideriamo lasciare il contesto corrente per entrare nella definizione di uno dei metodi invocati.

Durante il debug dell'applicazione, Visual Studio mostra degli strumenti diagnostici che evidenziano eventuali criticità prestazionali del nostro codice. Grazie a tali strumenti, riusciamo a scovare problemi che altrimenti resterebbero nascosti o, peggio, che si mostrerebbero solo in produzione, quando ormai l'applicazione è in uso da molti utenti contemporaneamente.

Debug con Visual Studio Code

Visual Studio Code è un editor molto snello, che ben si adatta a una grande varietà di linguaggi e tecnologie. Il debugger per le applicazioni .NET Core non è quindi incluso nel pacchetto di installazione ma va ottenuto come estensione gratuita dal Visual Studio Marketplace. Il nostro primo passo è dunque quello di accedere al riquadro delle estensioni di Visual Studio Code e installare l'estensione C# prodotta da Microsoft. Il debug di applicazioni ASP.NET Core scritte in F# o VB.NET non è al momento supportato da Microsoft. La Figura 2.14 mostra il riquadro dedicato alle estensioni.

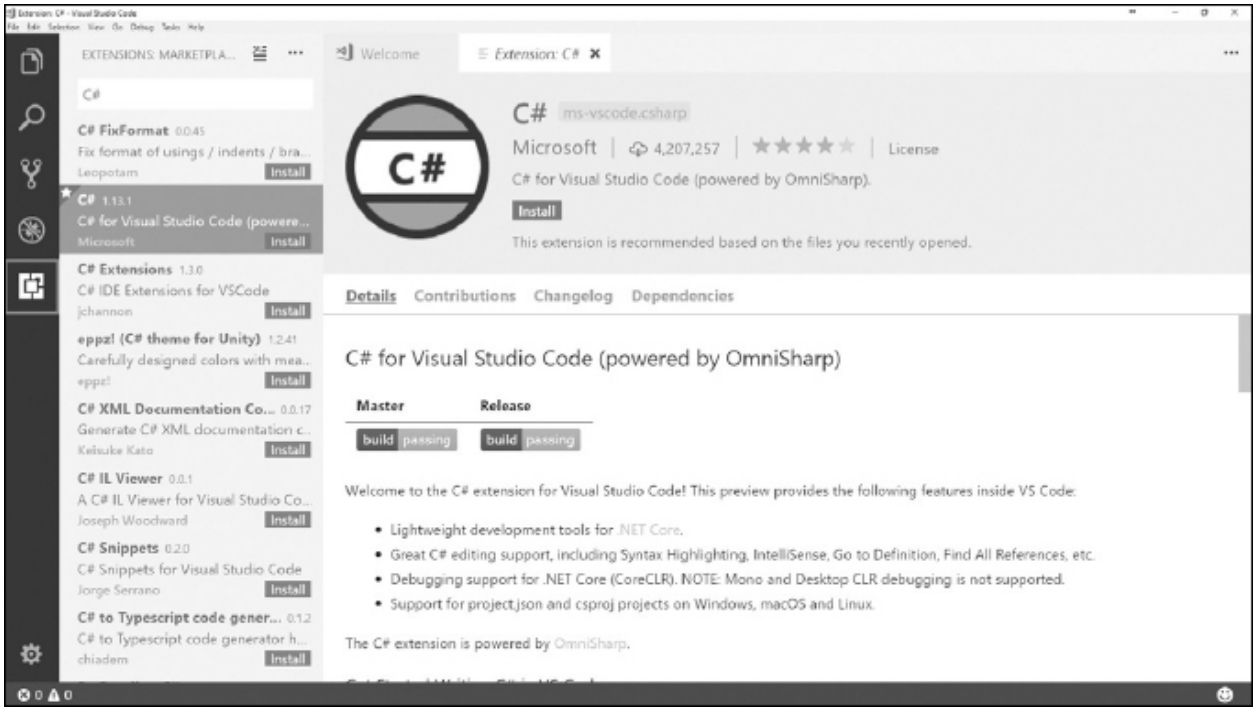


Figura 2.14 – Installazione dell'estensione C# da Visual Studio Code.

Una volta terminata l'installazione, clicchiamo il menu `File > Open Folder` per aprire la directory in cui era stato creato il progetto ASP.NET Core. Dopo pochi secondi, Visual Studio Code inizierà a scaricare i componenti necessari, incluso il .NET Core Debugger. L'operazione si completerà entro pochi minuti. Al termine, Visual Studio Code chiederà di creare la configurazione necessaria ad avviare il debug del progetto, come si può vedere nella Figura 2.15. Clicchiamo "Yes" per completare la procedura.

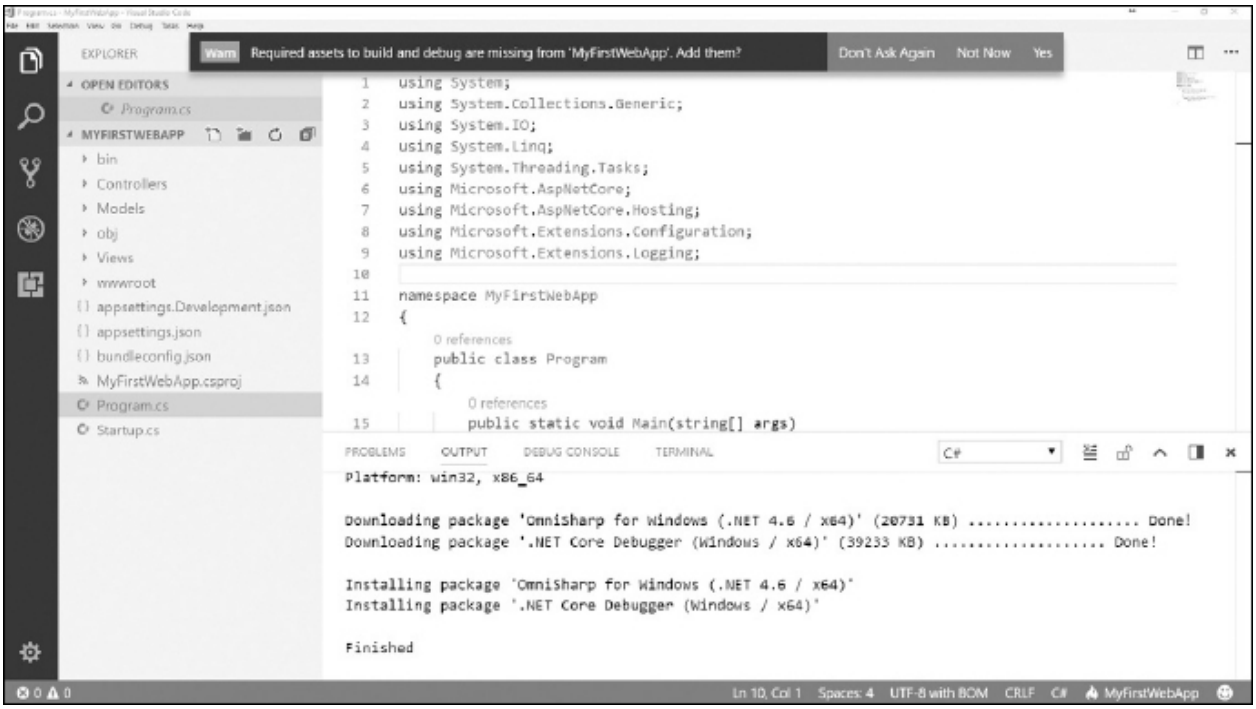


Figura 2.15 – Aggiungere la configurazione necessaria affinché il progetto possa essere avviato in debug.

A questo punto possiamo avviare il debug dall'apposita sezione, cliccando il bottone ".NET Core Launch (web)" o premendo il tasto F5 sulla tastiera. Anche in questo caso abbiamo l'opportunità di porre dei breakpoint in corrispondenza delle righe di codice. Quando l'esecuzione è in pausa, ispezioniamo il contenuto delle variabili dal riquadro dedicato al debug, visibile nella Figura 2.16.

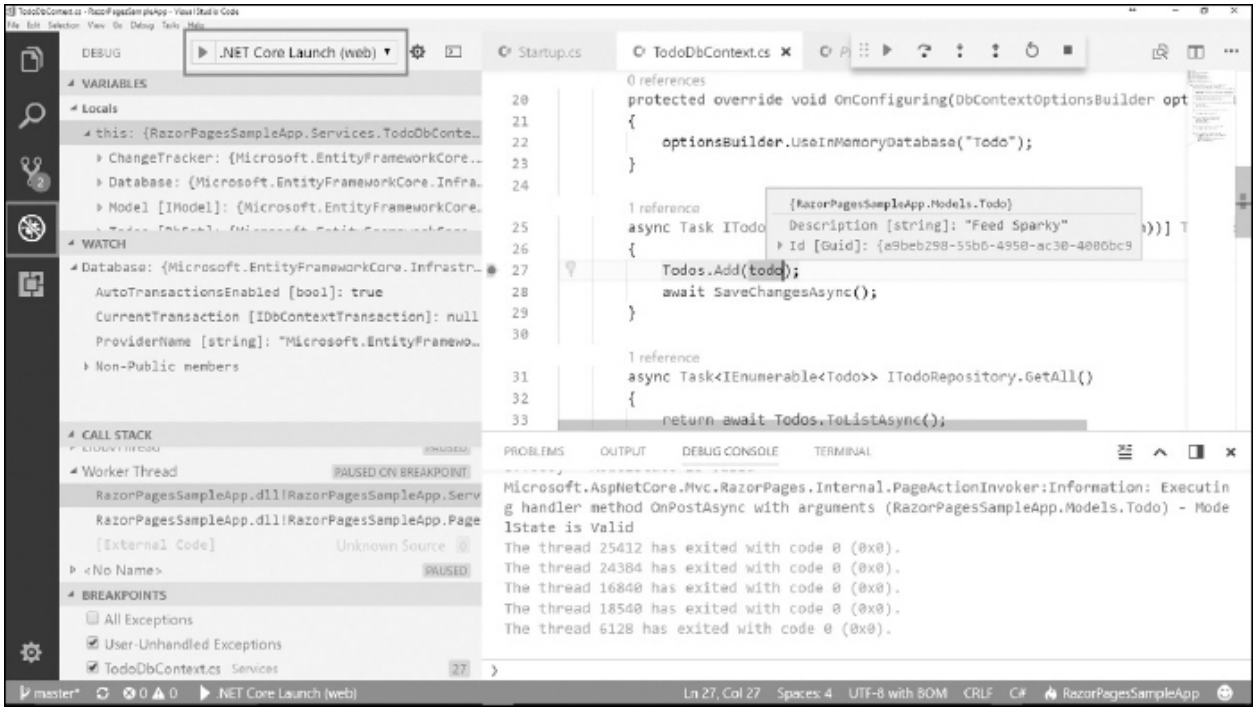


Figura 2.16 – Debug di un'applicazione ASP.NET Core con Visual Studio Code.

Durante il debug possiamo usare le consuete scorciatoie da tastiera per eseguire l'istruzione corrente (tasto F10), addentrarci in un metodo (tasto F11) o proseguire fino al breakpoint successivo (tasto F5).

Debug con Visual Studio for Mac

Visual Studio for Mac è ideale per lo sviluppo di applicazioni .NET Core e Xamarin su sistema operativo macOS. Trattandosi di un prodotto ancora giovane, non possiede ancora ogni strumento di diagnostica presente in Visual Studio ma offre comunque un'eccellente esperienza di debug. Proprio come nella controparte Windows, ci viene messo a disposizione un tasto "play" nella barra degli strumenti per avviare il debug dell'applicazione.

Durante il debug, ritroviamo tutti i riquadri fondamentali all'identificazione di possibili problemi.

- Il riquadro Locals, che mostra le variabili esistenti nel contesto attuale, con i relativi valori. Nel caso volessimo esaminare il valore di un'espressione più complessa, possiamo digitarla all'interno dello specifico riquadro Watch.
- Il riquadro Call Stack ci mostra l'attuale pila di invocazioni ai metodi, utile a capire quale sia stata la gerarchia di chiamate che ha condotto il programma ad arrestarsi al breakpoint corrente. Inoltre, il riquadro Threads mostrerà i vari percorsi di esecuzione attivi al momento.
- Il riquadro Immediate offre un'ulteriore opportunità per esaminare il valore restituito dalle nostre espressioni. Digitando direttamente in questo riquadro, possiamo sperimentare con il codice prima ancora di apportare modifiche al sorgente.

La Figura 2.17 offre una visione d'insieme dell'interfaccia durante il debug.

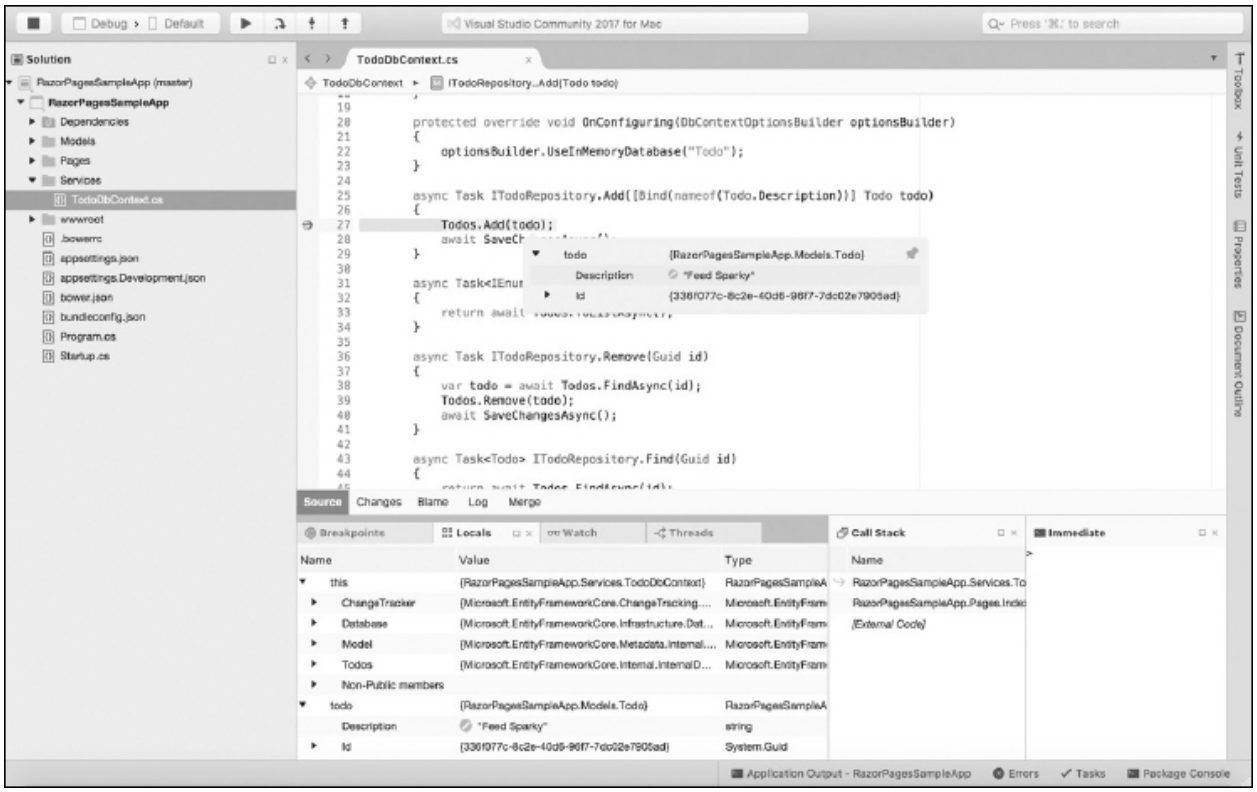


Figura 2.17 – Debug di un'applicazione ASP.NET Core da Visual Studio for Mac.

Visual Studio offre un'esperienza di debug molto soddisfacente, sia con singoli progetti sia con soluzioni multi-progetto. Di conseguenza, abbiamo la completa libertà di decidere come strutturare la nostra applicazione affinché sia organizzata al meglio.

Creare una solution con molteplici progetti

Man mano che un'applicazione cresce, diventa importante organizzarla in componenti più semplici, per evitare che diventi monolitica e perciò ingestibile. In alcuni casi, è addirittura consigliabile spezzarla in varie applicazioni dalle responsabilità specifiche. Si pensi, per esempio, a un sito e-commerce: anziché costruirlo come un blocco unico, potremmo realizzarlo facendo collaborare un'applicazione web rivolta agli acquirenti, una per la fatturazione e una per la gestione delle spedizioni. Dato che la suddivisione di un problema in parti più piccole è uno strumento così potente per affrontare la complessità, dovremmo tenere in considerazione l'idea di creare solution composte di molteplici progetti.

Creare la solution

Il primo passo consiste nel creare la solution, ovvero un file `.sln` che ha lo scopo di mantenere coesi i vari progetti. Grazie al .NET Core CLI, possiamo fare questa operazione da riga di comando:

Esempio 2.16

```
dotnet new sln
```

Tipicamente, il file `.sln` viene posto in una directory principale a cui poi verranno aggiunte altre sottodirectory, una per ogni progetto facente parte della solution.

Proviamo dunque a creare una nuova applicazione ASP.NET Core MVC, come abbiamo già visto in precedenza, e quindi ad aggiungerla come primo progetto della solution. I due comandi da lanciare saranno i seguenti:

Esempio 2.17

```
dotnet new mvc --output MyWebApp
dotnet sln add ./MyWebApp/MyWebApp.csproj
```

Ciascun progetto, che si tratti di un'applicazione ASP.NET Core o di altro tipo, possiede un file `.csproj` che ne descrive la struttura. Nell'esempio precedente, lo abbiamo aggiunto alla solution usando il suo percorso relativo.

Aggiungere un progetto di tipo Class Library

Spesso capita che più applicazioni debbano avvalersi di logica o strutture dati comuni. Per evitare di duplicare il codice, possiamo creare un progetto di tipo Class Library, ovvero un progetto che verrà referenziato da ogni applicazione interessata a usare il suo contenuto.

Pensiamo, per esempio, a un'applicazione mobile realizzata con Xamarin che debba visualizzare un catalogo di prodotti esposto da un'applicazione ASP.NET Core Web API. L'eventuale classe `Prodotto`, in questo caso, potrebbe essere definita in un progetto Class Library referenziato da entrambe. Ciò è possibile perché, nonostante siano applicazioni di tipo diverso, sia .NET Core che il runtime di Xamarin possono avvalersi di progetti Class Library conformi alla specifica .NET Standard, introdotta nel capitolo precedente e trattata in maniera approfondita nel Capitolo 23.

Per creare un progetto di tipo Class Library, usiamo ancora una volta il comando `dotnet new`. Dopo averlo aggiunto alla solution, facciamo in modo che venga anche referenziato dall'applicazione ASP.NET Core, affinché possa impiegare le classi definite al suo interno.

Esempio 2.18

```
dotnet new classlib --output MyLib
dotnet sln add ./MyLib/MyLib.csproj
dotnet add ./MyWebApp/MyWebApp.csproj reference ./MyLib/MyLib.csproj
```

Così, grazie al comando `dotnet add reference`, possiamo costruire relazioni tra i vari progetti.

Aggiungere un progetto di unit testing

Sottoporre il proprio codice a test automatici è sempre una buona pratica. Serve a confermarci che la nostra implementazione sta funzionando secondo la specifica e ci permette di aggiornare l'applicazione con serenità, perché eventuali bug di regressione verrebbero identificati subito, semplicemente mandando in esecuzione la suite di test.

Il .NET Core SDK è stato costruito per essere modulare e testabile; perciò offre il supporto a due framework di unit testing multipiattaforma: `MSTest` e `XUnit`. Le differenze tra i due framework – e un'introduzione più dettagliata alla pratica dello unit testing – le troviamo nella documentazione ufficiale, raggiungibile all'indirizzo: http://aspit.co/bke.

Supponendo di voler usare il framework `MSTest`, creiamo il relativo progetto e lo aggiungiamo alla nostra solution. Quindi, facciamo in modo che referenzi il progetto Class Library (ed eventualmente anche l'applicazione ASP.NET Core), affinché possa eseguire i test sulle classi definite in essa.

Esempio 2.19

```
dotnet new mstest --output MyTest
dotnet sln add ./MyTest/MyTest.csproj
dotnet add ./MyTest/MyTest.csproj reference ./MyLib/MyLib.csproj
```

A questo punto non resta che iniziare a scrivere i test all'interno del nuovo progetto. Quando siamo pronti per eseguirli, lanciamo questo comando:

Esempio 2.20

```
dotnet test ./MyTest/MyTest.csproj
```

In output vedremo apparire in verde il numero di test passati e in rosso quelli falliti.

Lavorare con i pacchetti NuGet

Presto o tardi avremo bisogno di integrare nella nostra applicazione delle funzionalità che non sono incluse in .NET Core ma che possiamo ottenere grazie al gestore di pacchetti NuGet.

Un pacchetto NuGet è un modo conveniente per installare nel progetto librerie e strumenti sviluppati da Microsoft o da terze parti. Questo sistema è così conveniente e diffuso che .NET Core stesso è composto di librerie distribuite come pacchetto NuGet. Affronteremo l'argomento in maniera approfondita nel Capitolo 23 ma nel frattempo vediamo quali sono i comandi essenziali da conoscere.

Aggiungere pacchetti NuGet al progetto

Supponiamo che la nostra applicazione ASP.NET Core abbia lo scopo di produrre un documento PDF a fronte dell'ordine di un acquirente. Dato che .NET Core non include la funzionalità di creazione dei PDF, è necessario ottenere una libreria che sia in grado di assolvere a questo compito. Se non conosciamo il nome del pacchetto che potrebbe fare al caso nostro, possiamo cercarlo all'interno della galleria NuGet, accessibile dall'indirizzo http://aspit.co/bkg.

Cercando il termine "PDF", troveremo sicuramente `iTextSharp`, uno dei pacchetti più popolari della galleria. Installiamolo nel progetto con il seguente comando:

Esempio 2.21

```
dotnet add ./MyFirstWebApp/MyFirstWebApp.csproj package iTextSharp
```

Da questo momento riusciremo a usare le classi offerte dalla libreria `iTextSharp`. Ovviamente, trattandosi di software sviluppato da terze parti, dovremo fare riferimento alla documentazione offerta dall'autore, che spesso si trova linkata nella pagina NuGet del pacchetto.

Creare un nostro personale pacchetto NuGet

Durante l'attività lavorativa, gli sviluppatori tendono a creare un proprio set di funzionalità da riutilizzare in vari progetti, in modo da accelerarne lo sviluppo. In questi casi si può valutare di includere quelle funzionalità in uno o più pacchetti NuGet, in modo che siano facilmente referenziabili da altri progetti. Il pacchetto non deve essere necessariamente pubblicato nella galleria ufficiale, ma può anche risiedere in un nostro feed privato. Il comando per la creazione del pacchetto estrae i metadati – come il nome, la versione, l'autore e la descrizione – dal file `.csproj`.

Esempio 2.22

```
dotnet pack ./MyLib/MyLib.csproj
```

Preparare l'applicazione per la distribuzione

Fino a questo punto, abbiamo eseguito e testato l'applicazione sul nostro PC o Mac di sviluppo. Dopo varie iterazioni, quando l'applicazione raggiunge un certo grado di maturità, arriva il momento di "pubblicarla" (o "metterla in produzione"), ovvero copiarla in una macchina che agisce da server affinché gli utenti possano accedervi per visualizzare pagine web e consumare i servizi che espone.

Questa attività di pubblicazione può presentare degli ostacoli di cui dovremmo essere consapevoli:

- È altamente probabile che la macchina server funzioni con un altro sistema operativo rispetto al nostro o addirittura con un'architettura di processore completamente diversa. Si consideri che .NET Core può funzionare anche su dispositivi con processori ARM a basso consumo elettrico, largamente impiegati nel mondo della Internet of Things. Per questo, è importante saper usare il .NET Core CLI per compilare l'applicazione in modo che funzioni sulla macchina server di destinazione (definita cross-compilazione).
- Se non abbiamo accesso diretto alla macchina server, non possiamo assicurarci che il .NET Core Runtime sia stato precedentemente installato dall'amministratore di quel server. In questa situazione, possiamo preparare la nostra applicazione in modo che contenga essa stessa il .NET Core Runtime, ovvero tutti gli assembly che le servono per funzionare (definita pubblicazione self-enclosed).
- È prassi comune che l'accesso alle applicazioni ASP.NET Core avvenga attraverso un web server robusto e ricco di funzionalità come IIS (solo per Windows), nginx o Apache (tipicamente usato su Linux). Il ruolo di questi web server consiste nel ricevere richieste web e inoltrarle all'applicazione ASP.NET Core, eventualmente alleviandone il carico grazie a servizi di cache o di trasferimento dei file statici (questo comportamento è definito reverse proxy server). La messa in produzione richiederà la creazione di un file di configurazione che descriva il modo in cui l'applicazione ASP.NET Core può essere raggiunta.

Il comando dotnet publish

Quando siamo pronti a pubblicare la nostra applicazione, lanciamo il comando `dotnet publish`. Tale comando, a differenza di quanto si potrebbe pensare, non serve a trasferire l'applicazione nella macchina server ma a preparare in una directory locale i file che andranno poi trasferiti manualmente. Da riga di comando, posizioniamoci nella directory principale del progetto e digitiamo quanto segue, eventualmente indicando il parametro `--output` per scegliere la directory di destinazione.

Esempio 2.23

```
dotnet publish --output Publish
```

Entrando nella directory Publish, troveremo un elenco di file come è illustrato nella Figura 2.18.

Figura 2.18 – Contenuto della directory di pubblicazione.

Nel caso ideale in cui il nostro PC di sviluppo abbia un sistema operativo compatibile con quello della macchina server (per esempio: Windows 10 e Windows Server 2016 a 64 bit), andiamo semplicemente a trasferire il contenuto della directory `Publish` nel server di produzione.

Usare il comando `dotnet publish` è il modo corretto per ottenere lo stretto necessario da copiare nel server. I file di codice sorgente non verranno assolutamente inclusi nella directory di pubblicazione e questo ci assicura che non verranno divulgati inavvertitamente. Questa modalità di pubblicazione è l'unica consigliata per ASP.NET Core e ciò rappresenta un miglioramento rispetto alle applicazioni ASP.NET tradizionali, che consentivano anche la pubblicazione dei file sorgenti con la modalità XCopy.

Cross-compilazione e pubblicazione self-contained

Nel caso in cui l'applicazione dovesse funzionare su una macchina server con sistema operativo o architettura di processore incompatibili con i nostri, è necessario rieseguire il comando `dotnet publish` indicando un runtime idoneo alla piattaforma di destinazione. Questa operazione è chiamata cross-compilazione e permette a un sistema di produrre un eseguibile valido per un sistema del tutto diverso. Supponiamo di aver sviluppato un'applicazione da un PC Windows e di voler pubblicare l'applicazione su un server Linux Ubuntu a 64 bit. In questo caso lanciamo il seguente comando, indicando l'opzione `--runtime ubuntu-x64` per indicare la piattaforma di destinazione.

Esempio 2.24

```
dotnet publish --runtime ubuntu-x64 --output Publish
```

Il valore `ubuntu-x64` è chiamato runtime identifier ed è soltanto uno dei numerosi identificatori disponibili per le varie piattaforme. Nuovi identificatori vengono aggiunti da Microsoft a ogni nuovo rilascio di .NET Core, man mano che il supporto viene esteso a nuove piattaforme. La gerarchia completa dei runtime identifier è disponibile nella documentazione ufficiale, raggiungibile all'indirizzo: http://aspit.co/bkh. Quando eseguiamo la cross-compilazione, ci accorgiamo che la directory `Publish` contiene anche gli assembly di .NET Core e le librerie specifiche per la piattaforma indicata, come si può vedere nella Figura 2.19. Questo comportamento è denominato pubblicazione self-contained e si verifica per default ogni volta che usiamo l'opzione `--runtime`.

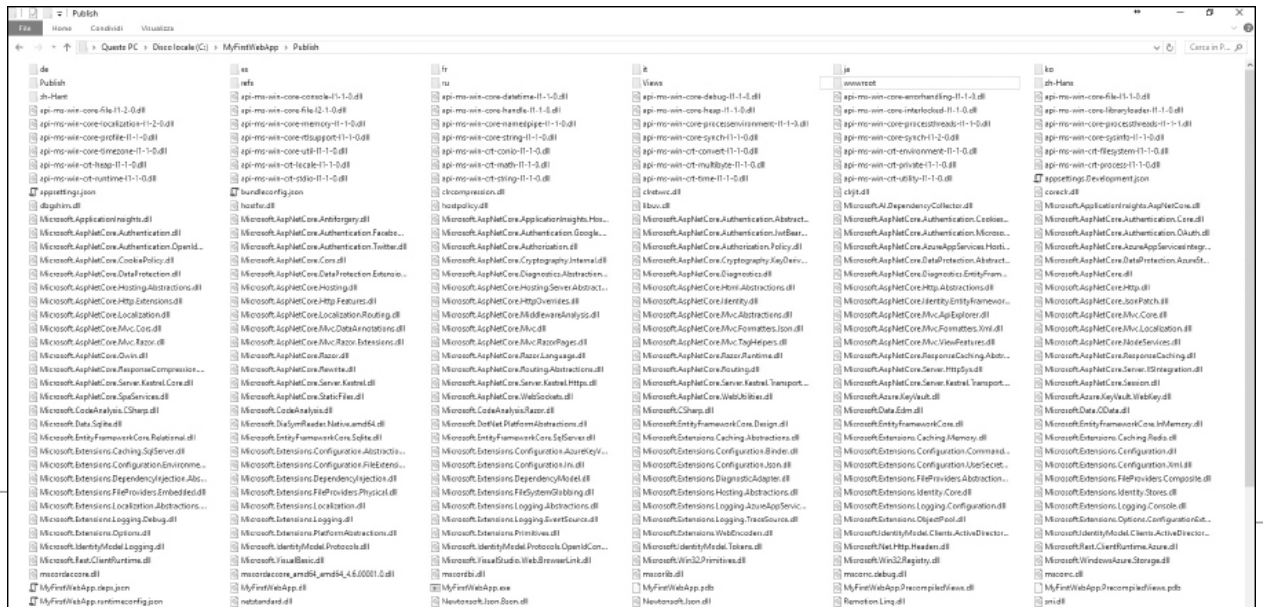


Figura 2.19 – La pubblicazione self-contained contiene tutto ciò che serve all'applicazione per funzionare.

La pubblicazione self-contained ci permette di avere in un'unica directory tutto ciò che serve all'applicazione per funzionare. Di conseguenza, non dovremo chiedere all'amministratore del server di pre-installare una certa versione del .NET Core a livello di sistema, dato che ora viene veicolato insieme all'applicazione. Questa è una buona soluzione che ci svincola dalle politiche di aggiornamento dell'hosting provider e ci rende completamente liberi di usare il runtime che preferiamo. Ovviamente, bisogna anche considerare una maggiore dimensione della directory di pubblicazione (circa 90 MB in più) che potrebbe rappresentare un problema solo nel caso di un'elevata densità di applicazioni esistenti sullo stesso server. Il numero di file, di per sé, non deve spaventarci: è solo un indizio dell'estrema modularità di .NET Core, che ci consente di costruire applicazioni snelle, formate solo dai componenti di cui necessitiamo.

Se decidiamo di fare a meno della pubblicazione self-contained, possiamo disabilitarla esplicitamente con l'opzione `--self-contained false`, come si vede nel comando che segue:

Esempio 2.25

```
dotnet publish --runtime ubuntu-x64 --self-contained false --output Publish
```

In questo modo, manteniamo la facoltà di scegliere il runtime identifier di destinazione e di avere una directory dalla dimensione contenuta. Ovviamente, in questo caso il .NET Core Runtime dovrà essere stato preventivamente installato nella macchina server.

Creare uno store a livello di sistema dei pacchetti NuGet

Nell'ottica di realizzare un server a elevata densità di applicazioni, è preferibile che l'amministratore del server installi nel sistema ogni versione del .NET Core Runtime necessaria alle applicazioni presenti sulla macchina, in maniera simile a quanto avveniva con .NET Framework tradizionale.

A partire da .NET Core 2.0, è anche possibile creare una cache di pacchetti NuGet a livello di sistema (denominata "store"), in modo da ridurre ulteriormente la dimensione delle applicazioni e velocizzarne i tempi di avvio. Affinché si possa sfruttare efficacemente lo "store" di pacchetti, è necessaria la stretta collaborazione dell'amministratore del server e dello sviluppatore.

- L'amministratore di sistema decide quali pacchetti inserire nello store, preparando un file manifest.xml che viene elaborato dal comando `dotnet store`. I pacchetti verranno inseriti all'interno della sottodirectory .dotnet/store nella directory del profilo dell'utente, ma è possibile scegliere un percorso differente grazie all'opzione `--output`.
- Lo sviluppatore riceve il file `manifest.xml` dall'amministratore, e invoca il comando `dotnet publish` con l'opzione `--manifest` per indicare al compilatore quali siano i pacchetti che non devono essere inclusi nell'output.

Le informazioni dettagliate per creare uno store di pacchetti NuGet sono disponibili nella documentazione online raggiungibile all'indirizzo: http://aspit.co/bki.

Avviare l'applicazione nella macchina server

Avendo pubblicato l'applicazione in una directory locale con il comando `dotnet publish`, non resta che trasferirne il contenuto nella macchina server. Per farlo, possiamo usare una varietà di tecniche, come il trasferimento FTP (per Windows, Linux e Mac), SCP o RSYNC (tipici del mondo Unix).

A trasferimento avvenuto, verifichiamo che l'applicazione riesca ad avviarsi lanciando il comando `dotnet` e fornendo come argomento il percorso dell'assembly principale, riconoscibile dal nome del progetto e dall'estensione dll. L'esempio seguente mostra il comando per l'avvio di un'applicazione ASP.NET Core denominata MyFirstWebApp.

Esempio 2.26

```
dotnet MyFirstWebApp.dll
```

In questo caso non ci è possibile avviare l'applicazione con il comando `dotnet run`, dato che il sottocomando `run` è disponibile solo con il .NET Core SDK e non con il .NET Core Runtime, consigliato per le installazioni su macchine server.

Anche se è certamente possibile avviare l'applicazione manualmente, come abbiamo appena fatto, è molto più verosimile che essa venga avviata dal web server che agisce da reverse proxy, tramite un proprio file di configurazione specifico. Avremo modo di rivisitare l'argomento nel Capitolo 22 ma, per ora, è sufficiente sapere che nella directory di pubblicazione troviamo il file `web.config` che rappresenta il file di configurazione di IIS per l'avvio dell'applicazione su server Windows.

Conclusioni

In questo capitolo abbiamo imparato a gestire progetti ASP.NET Core grazie al .NET Core CLI. Usare la riga di comando in maniera così estensiva può rappresentare una novità anche per gli sviluppatori .NET con più esperienza ma si tratta di un'abitudine da consolidare, dal momento che il .NET Core CLI è lo strumento indispensabile per lavorare su ogni piattaforma supportata.

Nel prossimo capitolo inizieremo a interessarci del progetto ASP.NET Core in sé, esaminando le sue parti costituenti per capirne il funzionamento e configurarlo in maniera precisa.

3
Anatomia di un progetto ASP.NET Core

Avendo esaminato il .NET Core CLI nel Capitolo 2, abbiamo tutti gli strumenti necessari per gestire il ciclo di vita dei nostri progetti ASP.NET Core. È il momento di dedicarci allo studio del progetto in sé, per avere una comprensione più profonda delle sue parti costituenti e di come poterle configurare secondo le nostre esigenze.

In questo capitolo impareremo a muoverci all'interno del progetto, iniziando a descrivere le directory e i file che lo compongono, fino a scoprire come interagiscono gli elementi della sua architettura. ASP.NET Core è un framework estremamente estendibile ma che, al tempo stesso, ci permette di iniziare con una configurazione di default molto concisa, che rende il progetto facile da comprendere anche a chi sta muovendo i primi passi nello sviluppo web.

Una visione d'insieme del progetto ASP.NET Core

Tenere il progetto ben organizzato è essenziale per assicurarci che resti gestibile anche al crescere della sua complessità. Il nostro compito sarà di gestire opportunamente alcuni tipi di contenuto:

- i file statici come gli stili CSS, i file JavaScript, le immagini e ogni altro contenuto multimediale;
- i file di codice che definiscono il comportamento lato server della nostra applicazione web;
- le fonti di configurazione che forniranno all'applicazione dei parametri di funzionamento.

Costruendo l'applicazione a partire da uno dei template offerti dal .NET Core CLI, possiamo già capire come organizzare al meglio i contenuti nel nostro progetto. La figura 3.1 illustra il contenuto tipico di un'applicazione ASP.NET Core MVC.

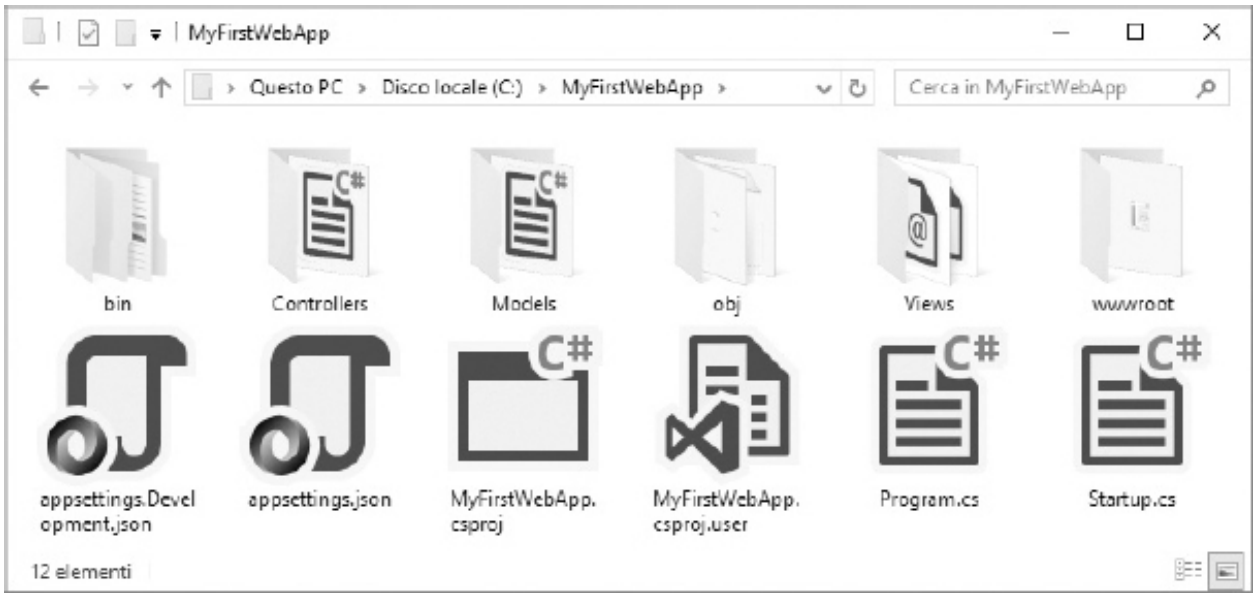


Figura 3.1 – File e directory che troviamo tipicamente in un'applicazione ASP.NET Core MVC.

Le directory del progetto

Sebbene ci venga concessa la libertà di organizzare i nostri contenuti in un numero arbitrario di directory, dobbiamo tenere a mente che alcune di esse acquisiscono uno scopo particolare all'interno dell'applicazione. Cominciamo con l'esaminare tali directory.

La directory wwwroot

La directory wwwroot è adibita a contenere i file statici dell'applicazione. In essa vanno posizionati tutti i contenuti che intendiamo rendere pubblicamente accessibili agli utenti, come gli stili CSS, le immagini, le librerie JavaScript e ogni altro tipo di documento. Per esempio, inserendo un'immagine `logo.jpg` all'interno della directory wwwroot, potrà essere visualizzata dall'utente digitando l'indirizzo `http://nomehost/logo.jpg` nel proprio browser. La directory wwwroot è perciò definita la web root del progetto e ogni file situato al di fuori di essa sarà di fatto inaccessibile mediante una richiesta HTTP. Il fatto che solo i file che si trovano nella web root sono scaricabili rappresenta un miglioramento alla sicurezza delle applicazioni ASP.NET Core. Infatti, si riducono i rischi che ogni altro tipo di file sia inavvertitamente reso pubblico via HTTP, come i file di log dell'applicazione o i PDF generati a partire da informazioni sensibili dei nostri utenti. Grazie a questo approccio opt-in, siamo maggiormente consapevoli di quali contenuti stiamo rendendo pubblici.

All'interno di wwwroot possiamo ovviamente creare sottodirectory per una migliore organizzazione. La figura 3.2 mostra un'immagine situata all'interno della directory `/wwwroot/gallery`.

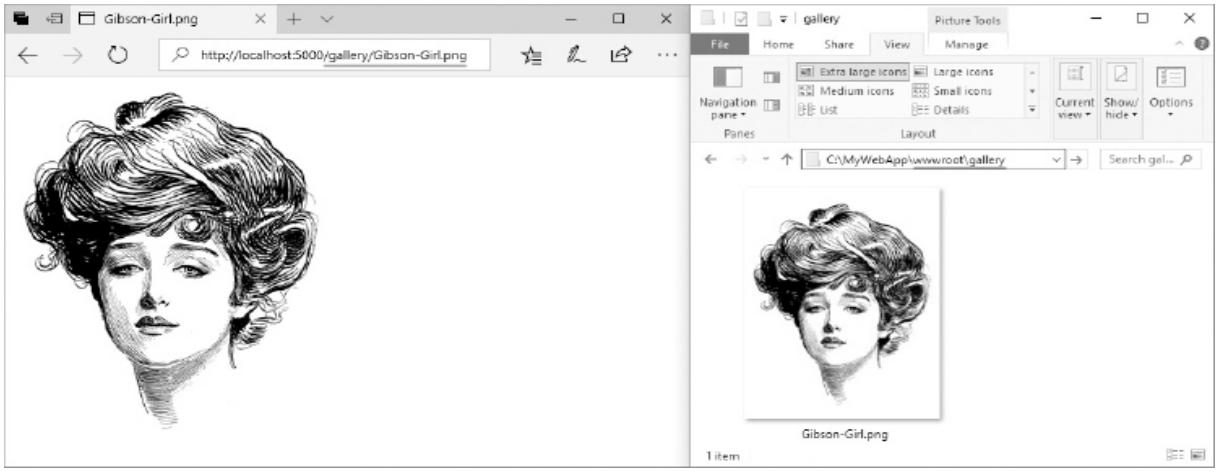


Figura 3.2 – Un'immagine creata in /wwwroot/gallery sarà raggiungibile dal percorso /gallery via HTTP.

A corredo del capitolo, viene fornita l'applicazione di esempio `gallery` che espone alcuni contenuti statici organizzati in varie sottodirectory di wwwroot.

Organizzare il codice in directory

Il vero centro nevralgico dell'applicazione è rappresentato dai file codice (C#, VB.NET o F#), che ci permettono di eseguire la logica per rispondere alle richieste HTTP degli utenti. Nel progetto ASP.NET Core possiamo creare un qualsiasi numero di sottodirectory dal nome arbitrario in cui inserire i file di codice. Questo è un buon modo per mantenere separati i componenti software che hanno responsabilità diverse all'interno dell'applicazione.

La figura 3.3 illustra un ipotetico file `CartManager.cs` creato all'interno di una cartella `/Services/Application`. Come unico accorgimento, la classe `CartManager` appartiene al namespace `MyWebApp.Services.Application`, la cui nomenclatura è coerente con la gerarchia di directory in cui il file si trova.

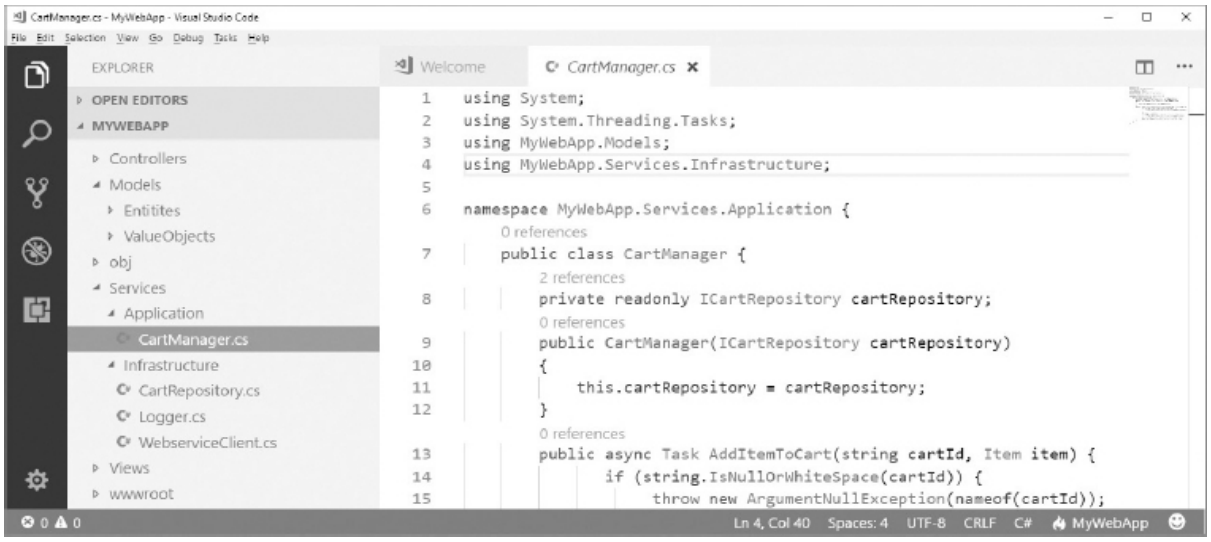


Figura 3.3 – I file di codice possono essere inseriti in directory dal nome arbitrario.

Nel corso del prossimo capitolo, trattando ASP.NET Core MVC, scopriremo come il codice rilevante per questo tipo di applicazioni sia organizzato nelle directory `Models`, `Views` e `Controllers`.

Le directory obj e bin

Compilando l'applicazione per la prima volta, nel nostro progetto appariranno le directory obj e bin.

- La directory obj contiene i cosiddetti intermediate object file, ovvero un insieme di file temporanei creati dal compilatore e necessari durante la seguente fase di compilazione;

- La directory bin contiene l'effettivo output di compilazione, ovvero gli assembly .dll che rappresentano il codice binario eseguibile della nostra applicazione.

All'interno di ciascuna di esse troveremo la sottodirectory `Debug` o `Release`, in base alla configurazione scelta all'atto della compilazione.

Durante lo sviluppo, le due cartelle `bin` e `obj` possono essere cancellate in qualsiasi momento e rigenerate con una nuova compilazione. Specie nel lavoro in team, quando si aggiunge il proprio codice sorgente a un repository TFVC o GIT, è preferibile escludere queste cartelle dal controllo di versione, in modo che il repository resti snello e privo di codice binario. Ogni sviluppatore che partecipa al progetto potrà facilmente rigenerarle automaticamente lanciando la compilazione.

La classe Program

Nel corso del Capitolo 2 abbiamo scoperto come le applicazioni ASP.NET Core siano anch'esse delle applicazioni console. Come si vede nell'esempio 3.1, infatti, la classe Program del progetto possiede il caratteristico metodo `Main` che è il punto d'ingresso dell'applicazione, ovvero il primo metodo a essere eseguito quando l'applicazione ASP.NET Core viene avviata.

Esempio 3.1

```
public class Program
{
   public static void Main(string[] args)
   {
      BuildWebHost(args).Run();
   }
   public static IWebHost BuildWebHost(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
         .UseStartup<Startup>()
         .Build();
}
```

Nel corpo del metodo `Main` viene preparato il cosiddetto web host, ovvero l'insieme dei servizi infrastrutturali che consentono all'applicazione di ricevere ed elaborare richieste HTTP. Il web host comprende almeno il web server e la pipeline dei middleware ma può accogliere anche altri servizi aggiuntivi come Application Insights, usato per la diagnostica di performance e utilizzo.

In ogni nuova applicazione noteremo che, nella classe `Program`, con pochissime righe di codice viene preparato un web host predefinito grazie al metodo `WebHost.CreateDefaultBuilder`.

Il web host è parte integrante nell'applicazione e perciò viene eseguito all'interno del suo stesso processo. La figura 3.4 mostra come interagisce con l'applicazione, con cui condivide un oggetto `HttpContext`, a rappresentare in maniera fortemente tipizzata la richiesta e la risposta corrente.

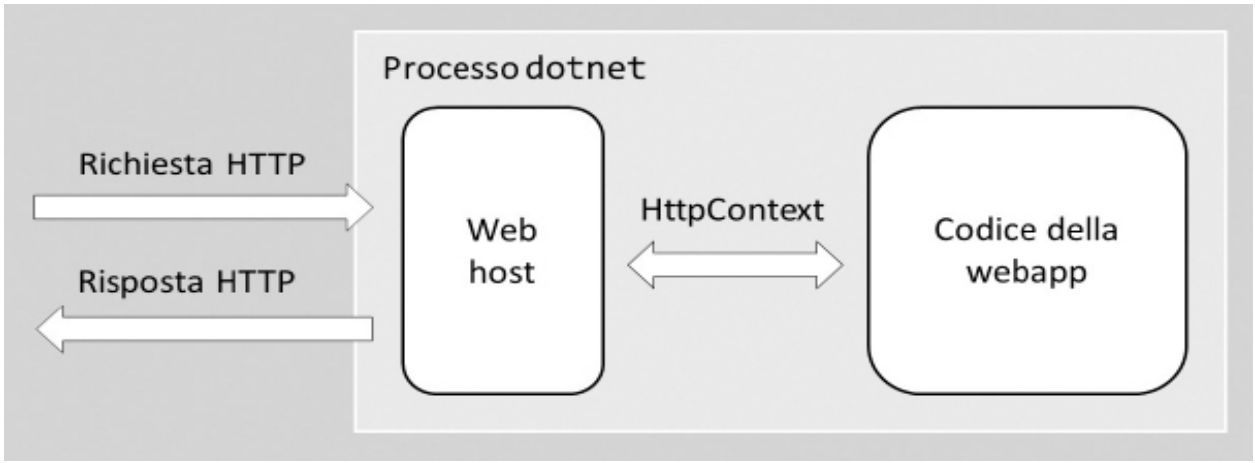


Figura 3.4 – Il web host e il codice dell'applicazione vivono nello stesso processo dotnet.

Il web host predefinito è certamente valido nelle prime fasi di sviluppo ma, prima o poi, potrebbe sorgere la necessità di configurarlo secondo le nostre esigenze. L'esatta configurazione di default del web host è consultabile nel repository GitHub del progetto, all'indirizzo http://aspit.co/bl0.

Personalizzare il web host

L'architettura di ASP.NET Core è estendibile e configurabile, a partire dal web host. In molte situazioni, è necessario intervenire, modificando alcuni suoi aspetti:

- gli endpoint, ovvero le interfacce di rete e le porte TCP su cui mettersi in ascolto di richieste;
- il web server e le sue opzioni specifiche, dal momento che ASP.NET Core può funzionare con diverse implementazioni che possiedono peculiarità differenti;
- il nome dell'environment, come "Development" o "Production", che avrà un effetto sul comportamento dell'applicazione in determinate situazioni, come quelle di errore;
- l percorso delle directory principali, proprio come la web root per i file statici di cui abbiamo già parlato in precedenza;
- la classe di configurazione dell'applicazione, uno dei punti più importanti in cui decidiamo quali middleware e servizi usare;
- l'integrazione con IIS, per esporre l'applicazione sul web in maniera sicura;
- l'uso di servizi di diagnostica come Application Insights, che possono aiutare sviluppatori e hosting provider a determinare quali siano i reali consumi dell'applicazione.

La configurazione degli aspetti citati può avvenire usando gli extension method dell'oggetto `WebHostBuilder`, che otteniamo invocando `WebHost.CreateDefaultBuilder`.

Inoltre, la configurazione del web host può avvenire usando fonti di configurazione esterna, in modo che non sia necessario ricompilare l'applicazione ogni volta che un valore deve essere modificato.

Indicare gli endpoint

Quando avviamo un'applicazione ASP.NET Core con il comando `dotnet run`, il web server si pone in ascolto sulla porta TCP 5000 dell'interfaccia di loopback (localhost), come risulta nell'Esempio 3.2.

Esempio 3.2

```
C:\MyWebApp> dotnet run

Hosting environment: Production
Content root path: C:\MyWebApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Questa configurazione di default può essere ovviamente modificata usando l'extension method `UseUrls` del `WebHostBuilder`. Forniamo al metodo uno o più argomenti nel formato URL, come abbiamo illustrato nell'esempio che segue.

Esempio 3.3

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .UseUrls("http://*:80", "http://localhost:5000", "http://10.0.0.3:8000")
      .UseStartup<Startup>()
.Build();
```

In questo caso, il web server si porrà in ascolto sulla porta TCP 80 di qualsiasi interfaccia di rete con indirizzo IPv4 o IPv6. In aggiunta, potrà ricevere richieste dalla porta TCP 5000 dell'interfaccia di loopback e dalla porta 8000 dell'interfaccia di rete a cui è stato assegnato l'indirizzo 10.0.0.3. Verifichiamo che l'impostazione abbia avuto successo osservando l'output che viene mostrato al riavvio dell'applicazione, come nell'Esempio 3.4.

Esempio 3.4

```
C:\MyWebApp> dotnet run

Hosting environment: Production
Content root path: C:\MyWebApp
Now listening on: http://[::]:80
Now listening on: http://localhost:5000
Now listening on: http://10.0.0.3:8000
Application started. Press Ctrl+C to shut down.
```

Quando scegliamo gli endpoint per la nostra applicazione, dobbiamo tener presente che non tutte le implementazioni dei web server hanno la robustezza necessaria per essere esposte direttamente su rete internet. Quando si usa il web server Kestrel (il default), Microsoft consiglia che sia posto in ascolto solo sull'interfaccia di loopback (localhost). Nel prossimo paragrafo esamineremo Kestrel e HTTP.sys, i due web server supportati da ASP.NET Core, e le relative modalità di deploy .

Scegliere un web server: Kestrel o HTTP.sys?

ASP.NET Core è stato concepito per comportarsi in maniera consistente su Windows, Linux e macOS. Per raggiungere questo obiettivo, Microsoft ha ripensato l'intero stack tecnologico — web server compreso — affinché fosse distribuibile con .NET Core su ogni piattaforma supportata.

Il web server Kestrel nasce per rispondere a questa esigenza ed è il web server in uso di default su ogni nuova applicazione ASP.NET Core creata da template. Una delle caratteristiche peculiari di Kestrel è la sua snellezza che lo rende estremamente efficiente nell'elaborare le richieste HTTP inviate dagli utenti.

A partire da ASP.NET Core 2.0, Kestrel ha raggiunto un buon grado di maturità e robustezza e perciò può essere esposto direttamente su Internet. In alternativa, può anche essere posto alle spalle di un reverse proxy che si occuperà di fornire servizi aggiuntivi come il load balancing, il caching dei contenuti o il trasferimento di file statici, così da alleviare il carico sull'applicazione ASP.NET Core.

I web server come IIS, Apache o NGINX possono agire da reverse proxy e inoltrare il traffico HTTP tra il client e Kestrel, come illustrato nella Figura 3.5. La scelta dipende ovviamente dalle preferenze personali e dalla piattaforma in uso. Usare un reverse proxy è anche un buon modo per esporre varie applicazioni ASP.NET Core sullo stesso nome host e per centralizzare i log delle richieste.

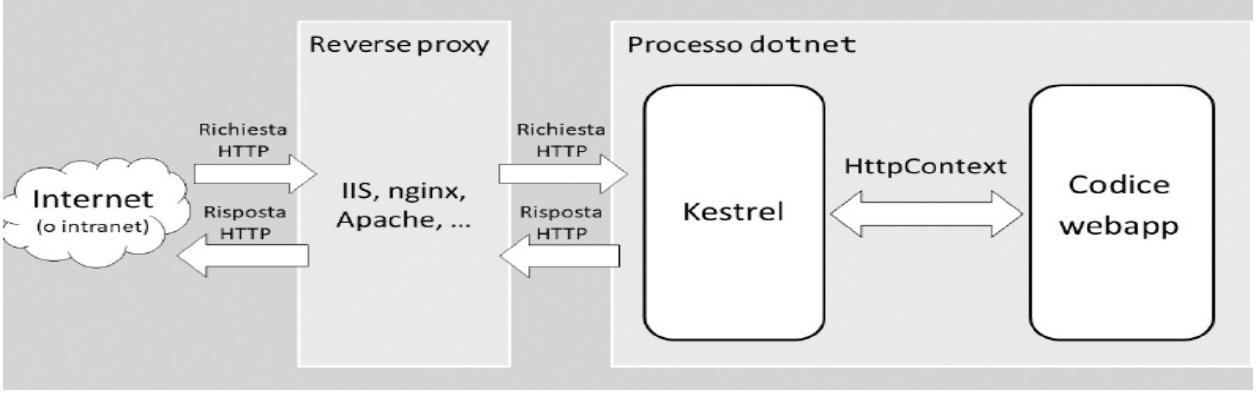


Figura 3.5 – Deploy di un'applicazione ASP.NET Core che usa Kestrel alle spalle di un reverse proxy.

Per configurare Kestrel, usiamo l'extension method `UseKestrel` durante la creazione del web host. In questo modo, potremo impostare alcuni aspetti specifici del web server, come la dimensione massima delle richieste o il numero di connessioni contemporanee consentite. L'Esempio 3.5 mostra l'impostazione di un limite massimo di 10MB per ogni richiesta HTTP.

Esempio 3.5

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .UseKestrel(options =>
      {
         //Usiamo l'oggetto options per configurare Kestrel
         //Ad esempio, limitiamo la dimensione della richiesta a 10 MB
         options.Limits.MaxRequestBodySize = 10 * 1024 * 1024;
      })
      //Abilitiamo l'integrazione con IIS, che agisce da reverse proxy
      .UseIISIntegration()
      .UseStartup<Startup>()
.Build();
```

Con l'extension method `UseIISIntegration` prepariamo Kestrel a integrarsi con IIS, in cui dovremo installare l'ASP.NET Core Module, necessario a completare l'integrazione. Le istruzioni per l'installazione sono riportate nella documentazione Microsoft all'indirizzo http://aspit.co/bl1.

Nel caso in cui le funzionalità offerte da Kestrel non fossero sufficienti nel nostro scenario, abbiamo l'opportunità di sostituirlo con HTTP.sys, ovvero lo stesso componente che è anche alla base di IIS.

Il web server HTTP.sys è disponibile solo su Windows ma fornisce ogni funzionalità che siamo già abituati a usare con IIS e con le applicazioni ASP.NET tradizionali. Alcuni dei motivi per cui potremmo preferirlo a Kestrel sono il suo supporto all'autenticazione Windows e la possibilità di esporre molteplici applicazioni sulla stessa porta TCP.

HTTP.sys si avvale dell'omonimo driver di Windows ed è un web server robusto e maturo, che può essere esposto direttamente su Internet, senza necessità di usare alcun reverse proxy, come illustrato nella Figura 3.6.

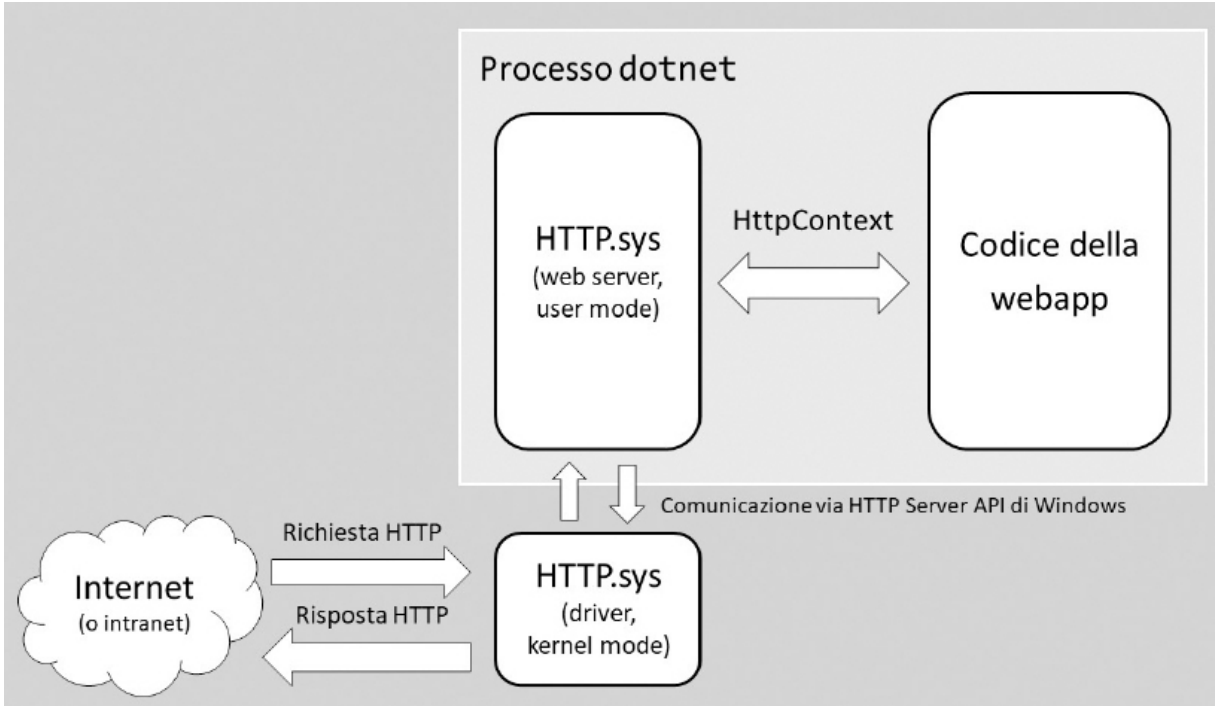


Figura 3.6 – Deploy di un'applicazione ASP.NET Core che usa il webserver HTTP.sys.

Se abbiamo la certezza che la nostra applicazione web verrà messa in produzione su Windows Server, usiamo l'extension method `UseHttpSys` per abilitare e configurare le opzioni di HTTP.sys. Nell'Esempio 3.6 usiamo HTTP.sys per sfruttare il suo supporto all'autenticazione Windows.

Esempio 3.6

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .UseHttpSys(options => {
         //Abilitamo l'autenticazione con account Windows
         options.Authentication.Schemes =
               AuthenticationSchemes.NTLM | AuthenticationSchemes.Negotiate;
         options.Authentication.AllowAnonymous = false;
      })
      .UseStartup<Startup>()
.Build();
```

Per scegliere più consapevolmente il web server più adatto alla nostra applicazione, la Tabella 3.1 mette a confronto le peculiarità offerte da entrambi.

Tabella 3.1 – Comparativa tra i web server Kestrel e HTTP.sys.

Caratteristica	Kestrel	HTTP.sys
Caratteristiche principali	Performance e portabilità	Robustezza e funzionalità
Multipiattaforma	Sì	Solo Windows
Open-source	Sì	Solo il webserver (non il driver)
Deploy e sicurezza	Da ASP.NET Core 2.0 può essere esposto su Internet	È robusto, può essere esposto su Internet
Può essere usato con IIS	Sì, con l'ASP.NET Core Module	No, ma non ne ha bisogno
Più domini sullo stesso IP (SNI)	Sì	Sì
Autenticazione Windows	No	Sì
HTTPS	Sì	Sì

HTTP/2	No, ma potrebbe arrivare con ASP.NET Core 2.2	Sì, a partire da Windows 10 e Windows Server 2016
WebSockets	Sì, con un middleware	Sì, a partire da Windows 8 e Windows Server 2012
Response caching	Sì, con un middleware o sfruttando il reverse proxy	Sì

Data l'estendibilità di ASP.NET Core, è possibile che in futuro si presentino implementazioni di web server di terze parti ad ampliare ulteriormente la scelta. Infatti, ASP.NET Core può funzionare con una qualsiasi implementazione dell'interfaccia IServer, registrabile con l'extension method `UseServer`.

Cambiare il percorso delle directory principali

In un'applicazione ASP.NET Core, il concetto di "directory principale" è distinto in due parti:

- La Content root è la directory in cui vengono cercati i contenuti dinamici dell'applicazione, come le view Razor di un'applicazione ASP.NET Core MVC situate nella sottodirectory View. Tratteremo approfonditamente questo aspetto nel corso del Capitolo 6. Per configurazione predefinita, è la directory principale del progetto, ovvero quella in cui si trovano anche il file .csproj e le classi `Program` e `Startup`;
- La Web root contiene i file statici e ogni file situato in essa è pubblicamente raggiungibile con una richiesta HTTP. Per configurazione predefinita è la directory wwwroot.

ASP.NET Core è abbastanza flessibile da permetterci di modificare i percorsi di entrambe le root directory usando gli extension method `UseContentRoot` e `UseWebRoot`. Come argomento, possiamo fornire percorsi assoluti o relativi, come risulta nell'Esempio 3.7.

Esempio 3.7

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .UseContentRoot(@"C:\mycontentroot")
      .UseWebRoot("wwwroot2")
      .UseStartup<Startup>()
.Build();
```

Impostare un environment

Le applicazioni web tengono spesso un comportamento differente durante le fasi di sviluppo e produzione. Consideriamo, per esempio, una situazione di errore imprevisto: mentre siamo in sviluppo, è preferibile che l'applicazione fornisca il maggior numero di dettagli tecnici possibile, per aiutarci a risolvere la problematica prontamente. In produzione, invece, l'applicazione deve fornire istruzioni più semplici e comprensibili per gli utenti, che li aiutino a capire come affrontare il problema.

La gestione degli errori è solo una tra le situazioni che comportano delle differenze. Altri esempi sono: l'uso della cache, il livello di logging, la diagnostica e così via.

ASP.NET Core ci permette di applicare una configurazione diversa in base all'environment in cui viene eseguita. Gli ambienti più comuni tra cui scegliere sono tre:

- Development è da usare durante lo sviluppo. In questo environment, l'applicazione è configurata per fornire molte informazioni diagnostiche, anche a scapito delle prestazioni.
- Staging è inteso per la fase di test che tipicamente precede l'entrata in produzione. L'applicazione viene configurata ed eseguita come se si trovasse già in produzione a eccezione di alcuni aspetti configurati in maniera differente, come le stringhe di connessione al database.
- Production è il default ed è il valore usato per l'environment di produzione. L'applicazione viene privata dei componenti superflui, in modo che sia più efficiente.

Per scegliere l'environment, usiamo l'extension method `UseEnvironment` durante la costruzione del web host, come abbiamo illustrato nell'Esempio 3.8.

Esempio 3.8

```
public static IWebHost BuildWebHost(string[] args) {
   return WebHost.CreateDefaultBuilder(args)
      //Possiamo scegliere il nome tra le costanti di EnvironmentName
      //oppure fornire una nostra stringa arbitraria (es. "Staging2")
      .UseEnvironment(EnvironmentName.Development)
      .UseStartup<Startup>()
      .Build();
}
```

Un modo alternativo consiste nell'impostare la variabile d'ambiente `ASPNETCORE_ENVIRONMENT`. Con Visual Studio, tale variabile può essere impostata facilmente dalla sezione "Debug" nelle proprietà del progetto, come si può notare nella figura 3.7.

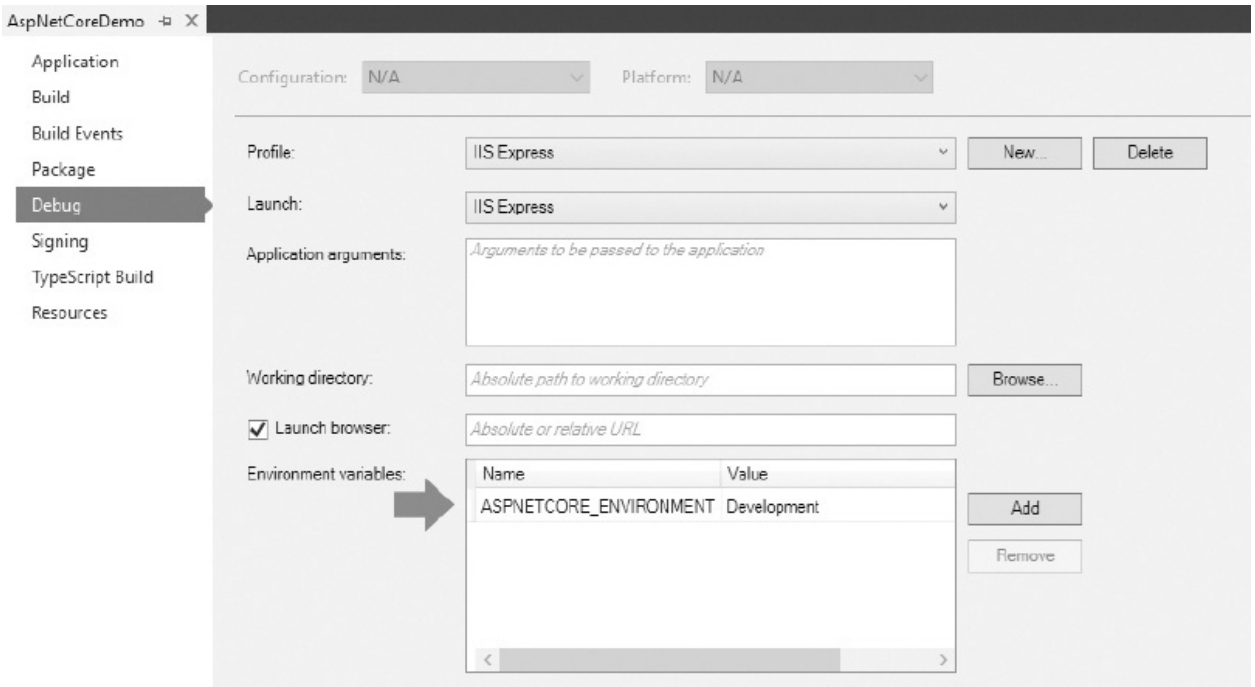


Figura 3.7 – Impostazione di ASPNETCORE_ENVIRONMENT dalle proprietà del progetto con Visual Studio.

Configurare il web host da fonti esterne

Finora abbiamo visto come sia possibile configurare il web host attraverso vari extension method. Sebbene questa tecnica sia efficace, ci costringe a ricompilare l'applicazione nel momento in cui un qualsiasi valore dovesse cambiare. Per ovviare a questo problema, ASP.NET Core supporta molte fonti di configurazione esterna, come i comuni file di testo (nei formati XML, JSON e INI), le variabili d'ambiente, i parametri da riga di comando e anche semplici oggetti in memoria. Inoltre, supporta fonti di configurazione che migliorano la riservatezza dei valori, come l'Azure Key Vault e gli User Secrets, di cui parleremo nel corso del Capitolo 18, dedicato alla sicurezza.

Per scegliere le fonti di configurazione, posizioniamoci nella classe `Program` e costruiamo un oggetto di tipo `ConfigurationBuilder`. Le varie fonti devono essere aggiunte in ordine di priorità (dalla meno importante alla più importante) usando il relativo extension method `Add`. L'ordine di aggiunta è fondamentale per determinare quale valore verrà assunto da ciascuna chiave di configurazione, dato che ognuno può essere definito (e ridefinito) nelle varie fonti.

L'Esempio 3.9 mostra come usare due fonti: un file `hosting.json` e le variabili d'ambiente con prefisso `ASPNETCORE_`, aggiunte dopo il file e perciò più prioritarie. L'oggetto così costruito viene fornito al web host grazie al metodo `UseConfiguration`.

Esempio 3.9

```
public static IWebHost BuildWebHost(string[] args) {

   var configuration = new ConfigurationBuilder()
      .SetBasePath(Directory.GetCurrentDirectory())
      .AddJsonFile("hosting.json", optional:true)
      .AddEnvironmentVariables("ASPNETCORE_")
      .Build();
   return WebHost.CreateDefaultBuilder(args)
```

```
        .UseStartup<Startup>()
        .UseConfiguration(configuration)
        .Build();
}
```

Secondo questa impostazione, il file `hosting.json` verrà cercato nella directory principale del progetto. Al suo interno, definiamo una o più chiavi di configurazione come viene illustrato dall'Esempio 3.10.

Esempio 3.10

```
{
   "urls": "http://*:80;http://localhost:5000;http://10.0.0.3:8000",
   "environment": "Development",
   "webRoot": "wwwroot"
}
```

Per valorizzare le variabili d'ambiente, invece, seguiamo i comandi illustrati dalla tabella 3.2.

Tabella 3.2 – Impostazione di una variabile d'ambiente a livello di sistema.

Piattaforma	Comando
Windows	`setx ASPNETCORE_URLS "http://*:5000"` e riavviare il prompt dei comandi.
Linux	Eseguire quanto segue con l'account root:echo "ASPNETCORE_URLS=http://*:5000">>/etc/environment e riavviare il sistema.
macOS	`echo "export ASPNETCORE_URLS=http://*:5000">>~/.bash_profile` e riavviare il terminale.

In allegato a questo capitolo viene fornita l'applicazione di esempio `endpoint`, che dimostra come sia possibile configurare gli endpoint del web host a partire da varie fonti di configurazione esterna.

L'elenco esaustivo delle chiavi di configurazione supportate dal web host è riportato nella documentazione ufficiale, all'indirizzo: http://aspit.co/bl2.

Scegliere la classe di configurazione dell'applicazione

Tutti gli esempi mostrati finora erano rivolti alla configurazione del web host: gli endpoint, i percorsi e il web server sono aspetti che riguardano la parte infrastrutturale dell'applicazione.

Come fare, invece, per configurare il comportamento dell'applicazione stessa? Con ASP.NET Core, questa responsabilità è centralizzata in una classe specifica, che indichiamo con l'extension method `UseStartup` durante la costruzione del web host. Questo metodo è così essenziale che lo ritroviamo anche in configurazioni minimali, motivo per cui è apparso numerose volte negli esempi precedenti. Rivediamolo ancora nell'Esempio 3.11.

Esempio 3.11

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .UseStartup<Startup>()
.Build();
```

La classe `Startup` è usata da ASP.NET Core immediatamente dopo la preparazione del web host.

Configurare l'applicazione con la classe Startup

In ogni nuova applicazione ASP.NET Core è sempre presente la classe `Startup`, situata nella directory principale del progetto. Il suo scopo è quello di configurare due tipi di componenti diversi:

- i Middleware, ovvero le classi chiamate a elaborare ogni richiesta HTTP. Ogni middleware offre una funzionalità specifica e si trova disposto con gli altri in maniera seriale lungo una pipeline. Più middleware sono presenti, più la nostra applicazione si arricchisce di funzionalità, seppur con un conseguente aumento dei tempi di elaborazione della richiesta. I middleware usati nell'applicazione li troviamo nel metodo `Configure` della classe `Startup`;
- i Servizi, ovvero i componenti che vengono riutilizzati in uno o più punti della nostra applicazione. I servizi vengono aggiunti dal metodo `ConfigureServices` della classe `Startup` e, così facendo, deleghiamo ad ASP.NET Core la responsabilità di gestire il loro ciclo di vita. Tratteremo i servizi in maniera più approfondita nel Capitolo 5, parlando del meccanismo di dependency injection integrato in ASP.NET Core.

L'Esempio 3.12 presenta lo scheletro di una classe Startup con i due suoi metodi caratteristici per la configurazione di middleware e servizi.

Esempio 3.12

```
public class Startup
{
   public void ConfigureServices(IServiceCollection services)
   {
      //Qui aggiungiamo i servizi alla collezione services
   }
   public void Configure(IApplicationBuilder app, IHostingEnvironment env)
   {
      //Qui decidiamo quali middleware usare nella pipeline
   }
}
```

Usare middleware dal metodo Configure

ASP.NET Core adotta un approccio molto trasparente quando si tratta di configurare i middleware: tutto il controllo viene lasciato allo sviluppatore ed è egli stesso a decidere, in maniera molto precisa, quanti, quali e in che ordine debbano essere usati nella pipeline di elaborazione delle richieste HTTP.

Nulla viene dato per scontato: un'applicazione ASP.NET Core minimale, creata con il comando `dotnet new web`, è per lo più una tela vergine che non offre nemmeno funzionalità basilari come il dowload di file statici. È lo sviluppatore, aggiungendo middleware, a decretare quali saranno le funzionalità, e perciò le prestazioni, della sua applicazione.

L'Esempio 3.13 mostra il contenuto del metodo `Configure` della classe `Startup` di un'applicazione ASP.NET Core più interessante, creata con il comando `dotnet new mvc`. In questo esempio troviamo alcuni middleware di uso comune, ciascuno configurato con il relativo extension method `Use`.

Esempio 3.13

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   if (env.IsDevelopment())
   {
      //Middleware per la pagina di errore contenente dettagli tecnici
      //(Solo se l'environment è stato impostato su "Development")
      app.UseDeveloperExceptionPage();
   }
   else
   {
      //Middleware che in caso di errore reindirizza a un determinato url
      //(Solo se l'environment è diverso da "Development")
      app.UseExceptionHandler("/Home/Error");
   }
   //Middleware che aggiunge il supporto ai file statici
   app.UseStaticFiles();
   //Middleware di routing di ASP.NET Core MVC
   app.UseMvc(routes =>
   {
      routes.MapRoute(
         name: "default",
         template: "{controller=Home}/{action=Index}/{id?}");
   });
}
```

Il metodo `IsDevelopment` del servizio `IHostingEnvironment` ci aiuta a configurare la pipeline in maniera diversa, a seconda dell'environment usato. Aggiungere solo i middleware necessari avrà effetti positivi sulle prestazioni dell'applicazione, apprezzabili in special modo durante uno stress test.

Come funzionano i middleware

Ciascun middleware, che sia stato sviluppato da Microsoft, da terze parti o da noi stessi, assolve a una responsabilità ben specifica. Alcuni esempi sono: mostrare una pagina di errore, servire file statici, impedire l'accesso agli utenti anonimi, loggare una richiesta, fare il routing degli Url, e così via.

Ogni richiesta HTTP è gestita dal primo middleware della pipeline che, come ogni altro, ha facoltà di:

- Esaminare la richiesta e decidere se farla elaborare anche al middleware successivo o no.
- Produrre una risposta per il client.
- Alterare la richiesta, oppure alterare la risposta prodotta da un altro middleware.

La Figura 3.8 mostra una pipeline in cui sono stati configurati 4 middleware: i primi due esaminano la richiesta e lasciano che venga elaborata dal middleware successivo. Il terzo, invece, si occupa di produrre una risposta per il client, impedendo che la richiesta venga elaborata dal quarto middleware.

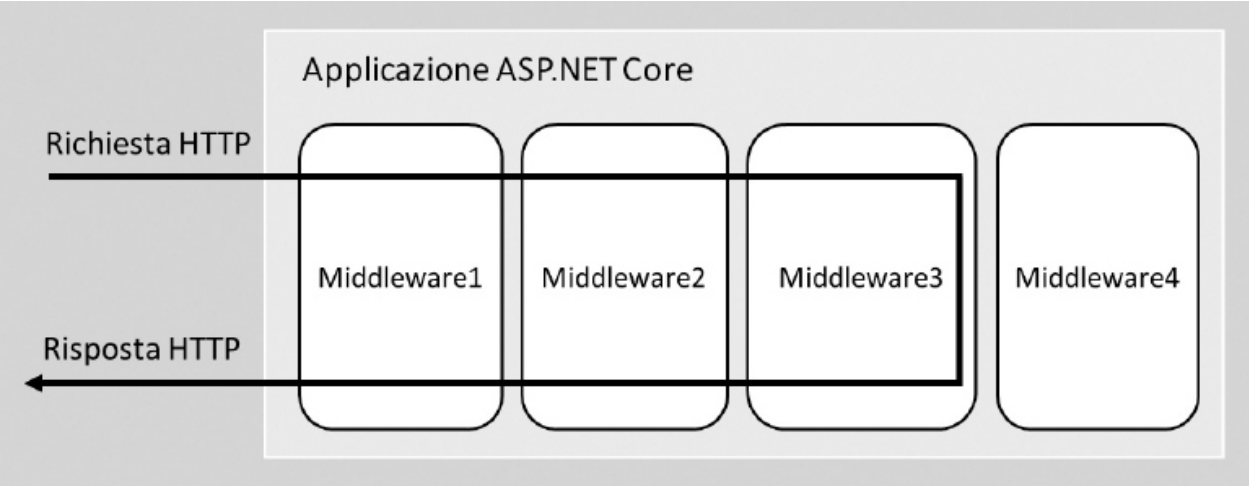


Figura 3.8 – I middleware scelgono se lasciar proseguire la richiesta al successivo o se fornire una risposta.

Si evince che l'ordine d'uso dei middleware è importante: il primo della pipeline sarà dunque il primo ad avere l'opportunità di esaminare la richiesta, ma anche l'ultimo a poter alterare la risposta. Affinché possano adempiere adeguatamente al loro scopo, alcuni middleware devono preferibilmente essere aggiunti all'inizio della pipeline. Alcuni esempi sono:

- Pagina di errore: questo middleware deve attendere che tutti i successivi abbiano compiuto la loro elaborazione per poter determinare se si è verificato un errore oppure no. Essere il primo della pipeline gli conferisce anche il privilegio di essere l'ultimo a poter alterare la risposta per fornire informazioni a proposito dell'errore.
- File statici: se il client sta richiedendo un file statico (per esempio: un'immagine JPG), per motivi prestazionali è preferibile che la richiesta non percorra tutta la pipeline, dato che il middleware dei file statici può esso stesso rispondere con il contenuto binario del file richiesto.
- Autorizzazione: anche in questo caso, è preferibile che la richiesta non percorra la pipeline se l'utente non possiede i privilegi per accedere alla risorsa richiesta (per esempio: la pagina di un'area riservata). In questo caso, il middleware di autorizzazione può proteggere l'applicazione, impedendo il proseguimento della richiesta. La sua risposta conterrà un messaggio di errore per informare l'utente che deve autenticarsi.

Nel corso del libro torneremo più volte nel metodo `Configure` per usare i middleware offerti da ASP. NET Core. L'elenco completo è disponibile nella documentazione, all'indirizzo: `http://aspit.co/bl3`.

Aggiungere servizi dal metodo ConfigureServices

Il metodo `ConfigureServices` della classe `Startup` è il responsabile per l'aggiunta dei servizi che intendiamo rendere disponibili agli altri componenti dell'applicazione. L'Esempio 3.14 mostra il contenuto di tale metodo in un'applicazione ASP.NET Core MVC.

Esempio 3.14

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
}
```

La collezione `IServiceCollection` rappresenta un catalogo al quale possiamo aggiungere servizi, che siano di terze parti o personalizzati. Tipicamente, per ogni servizio esiste anche un extension method Add che ne permette la configurazione.

Cos'è un servizio

Un servizio è una classe che svolge una singola responsabilità in maniera compiuta, e che possiamo riutilizzare in vari punti dell'applicazione per evitare la duplicazione di codice. Alcuni servizi possono occuparsi di compiti infrastrutturali, come salvare dati in un database o scrivere in un file di log. Altri, invece, si occupano di compiti applicativi, come calcolare il prezzo di vendita di un prodotto in base alla quantità acquistata oppure determinare se un particolare impiegato può prendere in prestito una determinata auto aziendale.

I servizi possono essere utilizzati in ogni componente dell'applicazione, middleware compresi. La Figura 3.9 mostra per l'appunto i middleware di una pipeline che si avvalgono di vari servizi per svolgere correttamente la propria mansione.

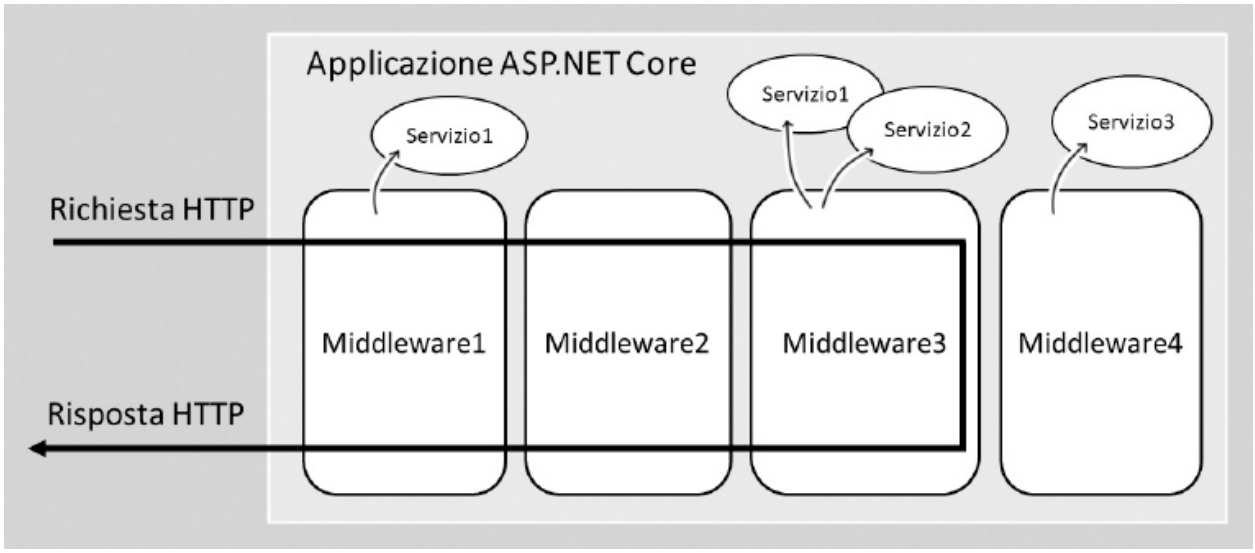


Figura 3.9 – I servizi possono essere usati da vari componenti dell'applicazione, come i middleware.

Quando un componente si avvale di un servizio, si dice che "dipende" da esso. Approfondiremo questo aspetto parlando di dependency injection nel Capitolo 5.

Configurare l'applicazione

Ogni applicazione web utilizza valori di configurazione per integrarsi con sistemi esterni: stringhe di connessione, parametri SMTP e chiavi API sono solo alcune delle informazioni che entrano in gioco durante il suo ciclo di vita. Per evitare che queste informazioni siano cablate nel codice, si preferisce definirle in fonti di configurazione esterna, in modo che possano essere modificate in qualsiasi momento, anche dopo il rilascio.

Per default, l'applicazione ASP.NET Core legge i valori di configurazione dal file `appsettings.json` che troviamo nella cartella del progetto, oltre che da eventuali variabili d'ambiente definite nel sistema. Possiamo cambiare questa impostazione predefinita grazie al `ConfigurationBuilder`, che ci permette di comporre a piacimento le varie fonti di configurazione supportate. Si tratta dello stesso approccio che abbiamo già usato per la configurazione del web host.

Variare la configurazione in base all'environment

Il meccanismo di configurazione di ASP.NET Core si dimostra particolarmente versatile quando vogliamo far variare i valori di configurazione in base all'environment.

Nell'Esempio 3.15 coinvolgiamo il servizio `IHostingEnvironment` per ottenere il nome dell'environment corrente e usarlo per rendere dinamico il percorso di un file JSON. Possiamo costruire la nostra configurazione nel costruttore della classe `Startup`.

Esempio 3.15

```
public class Startup
{
   public Startup(IHostingEnvironment en)
   {
      //Aggiungiamo le fonti di configurazione desiderate
      var builder = new ConfigurationBuilder()
         .SetBasePath(en.ContentRootPath)
         .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
         .AddJsonFile($"appsettings.{en.EnvironmentName}.json", optional: true)
         .AddEnvironmentVariables("ASPNETCORE_");
      Configuration = builder.Build();
   }
   //Questa proprietà mantiene un riferimento alla configurazione creata
   public IConfigurationRoot Configuration { get; }
   //I metodi Configure e ConfigureServices sono omessi per brevità
}
```

L'esempio precedente definisce tre fonti di configurazione: il file `appsettings.json`, un ulteriore file json opzionale, il cui nome dipende dall'environment, e le variabili d'ambiente. L'aver usato il valore di `EnvironmentName` ci permette di creare un file con un percorso dinamico, come per esempio `appsettings.Development.json`, che conterrà definizioni specifiche applicabili solo in quell'environment. Tale file, essendo stato aggiunto dopo `appsettings.json`, risulterà più prioritario rispetto a esso e perciò avrà la facoltà di ridefinirne i valori.

La Tabella 3.3 mostra un contenuto di esempio per i due file json. Dato che durante lo sviluppo si usa solitamente un database di test, il file `appsettings.Development.json` ridefinisce la connection string ma non le impostazioni Smtp, che invece resteranno invariate.

Tabella 3.3 – File base e file specifico per l'environment Development.

appsettings.json

```
{
   "ConnectionStrings": {
      "Default": "Server=MainSrv;
         Database=Db1; User Id=user1;
         Password=pass1"
   },
   "Smtp": {
      "Host": "smtp.example.com",
      "Port": 587,
      "EnableSsl": true,
      "Username": "user1",
      "Password": "pass1",
      "Email": "dev@example.com"
   }
}
```

appsettings.Development.json

```
{
   "ConnectionStrings": {
      "Default": "Server=TestSrv;
      Database=Test1; User Id=user1;
       Password=pass1"
   }
}
```

Mentre l'applicazione è in esecuzione nell'environment "Development", la connection string usata sarà quella rivolta al server `TestSrv` mentre, in tutti gli altri ambienti, il server usato sarà `MainSrv`, dato che il file `appsettings.Development.json` non verrebbe preso in considerazione.

Teniamo a mente che l'Esempio 3.15 impiega anche le variabili d'ambiente come fonte più prioritaria. Se definiamo una variabile d'ambiente di nome `ASPNETCORE_CONNECTIONSTRINGS:DEFAULT`, il suo valore andrebbe quindi a ridefinire quelli presenti nei due file json. Il simbolo dei due punti viene usato nei nomi delle variabili d'ambiente come carattere separatore nel percorso delle chiavi di configurazione. Possiamo notare che le chiavi di configurazione sono annidate a vari livelli di profondità, fatto che ci consente di organizzare i valori in sezioni, secondo una struttura gerarchica.

Leggere la configurazione in maniera fortemente tipizzata

Nell'Esempio 3.15 abbiamo creato una configurazione personalizzata basata su varie fonti. Ora, con l'Esempio 3.16, vediamo come usare il metodo `GetSection` per estrarre una particolare sezione dalla configurazione e fare in modo che i valori vengano resi disponibili in maniera fortemente tipizzata da una nostra classe.

Esempio 3.16

```
public void ConfigureServices(IServiceCollection services)
{
   services.Configure<SmtpConfiguration>(Configuration.GetSection("Smtp"));
}
```

Usando il metodo `services.Configure` abbiamo di fatto aggiunto ad ASP.NET Core il nostro primo servizio. Esso esporrà i valori di configurazione mediante una nostra classe personalizzata, in cui avremo definito proprietà idonee ad accogliere tali valori. L'Esempio 3.17 riporta la definizione della classe personalizzata `SmtpConfiguration`: possiamo notare come i nomi e i tipi delle proprietà siano coerenti con il contenuto della sezione "Smtp" definita nel file `appsettings.json`.

Esempio 3.17

```
public class SmtpConfiguration
{
   public string Host {get; set;}
   public int Port {get; set;}
   public bool EnableSsl {get; set;}
   public string Username {get; set;}
   public string Password {get; set;}
   public string Email {get; set;}
}
```

Nei punti in cui l'applicazione necessita di accedere alle impostazioni SMTP, dovrà avvalersi del servizio `IOptionsMonitor<SmtpConfiguration>`. L'Esempio 3.18 mostra un ipotetico middleware che usa le impostazioni SMTP per inviare e-mail di notifica degli errori.

Esempio 3.18

```
public class EmailErrorsMiddleware {
   private readonly SmtpConfiguration smtpConf;
   public EmailErrorsMiddleware(IOptionsMonitor<SmtpConfiguration> options)
   {
      //Le impostazioni attuali si trovano nella proprietà CurrentValue
      smtpConf = options.CurrentValue;
   }
}
```

Il servizio `IOptionsMonitor<T>` è in grado di monitorare alcuni tipi di fonte (come i file di testo) per rilevare i cambiamenti e ricaricare a caldo la nuova configurazione. I valori attuali vengono esposti attraverso la proprietà `CurrentValue` dell'oggetto. Questo ci evita di dover riavviare l'applicazione, massimizzando il suo uptime. In alternativa, possiamo ricorrere al servizio `IOptions<T>` che non supporta il monitoraggio delle fonti. In allegato a questo capitolo si trova l'applicazione di esempio `config`, che dimostra l'uso dei servizi di configurazione.

Che fine hanno fatto il web.config e il global.asax?

Il file web.config, nelle precedenti versioni di ASP.NET, era il principale file di configurazione dell'applicazione. Con ASP.NET Core, si è preferito adottare una soluzione più versatile, basata su molteplici e diversificate fonti di configurazione, come abbiamo visto nel corso del paragrafo precedente. Tuttavia, può ancora capitare di trovare un file web.config nelle applicazioni ASP.NET Core: esso infatti è ancora valido come file di configurazione del sito IIS. Se la nostra applicazione è ospitata su server Windows e usa IIS come reverse proxy, allora possiamo usare il nodo `system.webServer` del web.config per controllare alcuni aspetti di IIS, come definire delle regole di url rewriting o abilitare il caching dei contenuti. Al minimo, il web.config contiene la configurazione obbligatoria dell'ASP.NET Core Module che permetterà a IIS di avviare l'applicazione ASP.NET Core. L'Esempio 3.19 mostra tale configurazione.

Esempio 3.19

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.webServer>
      <handlers>
         <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
               resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet" arguments=".\MyWebApp.dll"
                     stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" />
   </system.webServer>
</configuration>
```

La guida all'utilizzo del nodo `system.webServer` è disponibile all'indirizzo http://aspit.co/bl4.

Il file global.asax consentiva invece di eseguire codice in corrispondenza di alcuni eventi particolari, come l'avvio dell'applicazione, momento ideale per configurare determinati componenti. Il file non è più necessario, dal momento che l'applicazione viene già configurata nei metodi `Configure` e `ConfigureServices` della classe `Startup`.

Inoltre, dato che ASP.NET Core si avvale di una pipeline estremamente modulare, non usa un ciclo di vita rigido e formato da una sequenza di eventi prefissati come nelle applicazioni ASP.NET "tradizionali". Se vogliamo introdurci nel ciclo di vita di una richiesta HTTP, dobbiamo semplicemente scrivere un middleware personalizzato e inserirlo opportunamente nella pipeline.

Il file .csproj

Il file con estensione .csproj continua a essere il file di progetto anche nelle applicazioni ASP.NET Core, seppure abbia ricevuto revisioni nella sua struttura. Esso contiene i metadati, l'elenco delle dipendenze e, in generale, ogni tipo di disposizione che permetta la corretta compilazione dell'applicazione da parte di MSBuild. L'Esempio 3.20 mostra il contenuto del file .csproj di una nuova applicazione ASP.NET Core.

Esempio 3.20

```
<Project Sdk="Microsoft.NET.Sdk.Web">
   <PropertyGroup>
      <TargetFramework>netcoreapp2.1</TargetFramework>
   </PropertyGroup>
   <ItemGroup>
      <Folder Include="wwwroot\" />
   </ItemGroup>
   <ItemGroup>
      <PackageReference Include="Microsoft.AspNetCore.App" Version="2.1.0" />
   </ItemGroup>
</Project>
```

Il codice contenuto nel file .csproj è decisamente conciso: oltre al moniker `netcoreapp2.1` che denota un'applicazione rivolta al .NET Core 2.1, troviamo solo i riferimenti al metapacchetto NuGet `Microsoft.AspNetCore.App` e alla directory wwwroot, i cui contenuti verranno inclusi all'atto della pubblicazione. Nessuna menzione viene fatta delle classi `Program` e `Startup` poiché, nella nuova struttura di progetto, ogni file di codice è già candidato alla compilazione. Solo nel caso in cui volessimo escludere un file di codice o modificare le opzioni di pubblicazione di un contenuto statico, ne troveremmo traccia nel file .csproj.

Il metapacchetto Microsoft.AspNetCore.App

Tutte le funzionalità offerte da ASP.NET Core sono state organizzate da Microsoft in una moltitudine di pacchetti NuGet. Questa grande suddivisione in componenti non rappresenta un problema e, anzi, ci permette di usare e distribuire solo le effettive funzionalità di cui necessitiamo. Per evitare che ogni nuovo progetto contenesse decine di riferimenti a pacchetti NuGet, è stato creato il metapacchetto `Microsoft.AspNetCore.App` che non include logica ma ha l'unico scopo di referenziare i vari altri pacchetti che potrebbero servirci per la nostra web application. Grazie a esso, nel file .csproj troveremo il riferimento a un solo pacchetto NuGet, così da restare molto conciso e di facile lettura.

Se vogliamo riguadagnare un controllo più granulare sui pacchetti referenziati dalla nostra applicazione, possiamo ovviamente rimuovere il riferimento a `Microsoft.AspNetCore.App` e aggiungere individualmente i pacchetti di cui necessitiamo. In questo modo, Microsoft offre un ingresso facilitato ad ASP.NET Core per agli sviluppatori principianti, senza rinunciare a un framework estremamente modulare, gradito agli sviluppatori più esperti e attenti alle performance.

L'elenco completo delle dipendenze di `Microsoft.AspNetCore.App` è disponibile nella pagina del pacchetto NuGet all'indirizzo http://aspit.co/bnr. La Figura 3.10 illustra alcune di tali dipendenze.

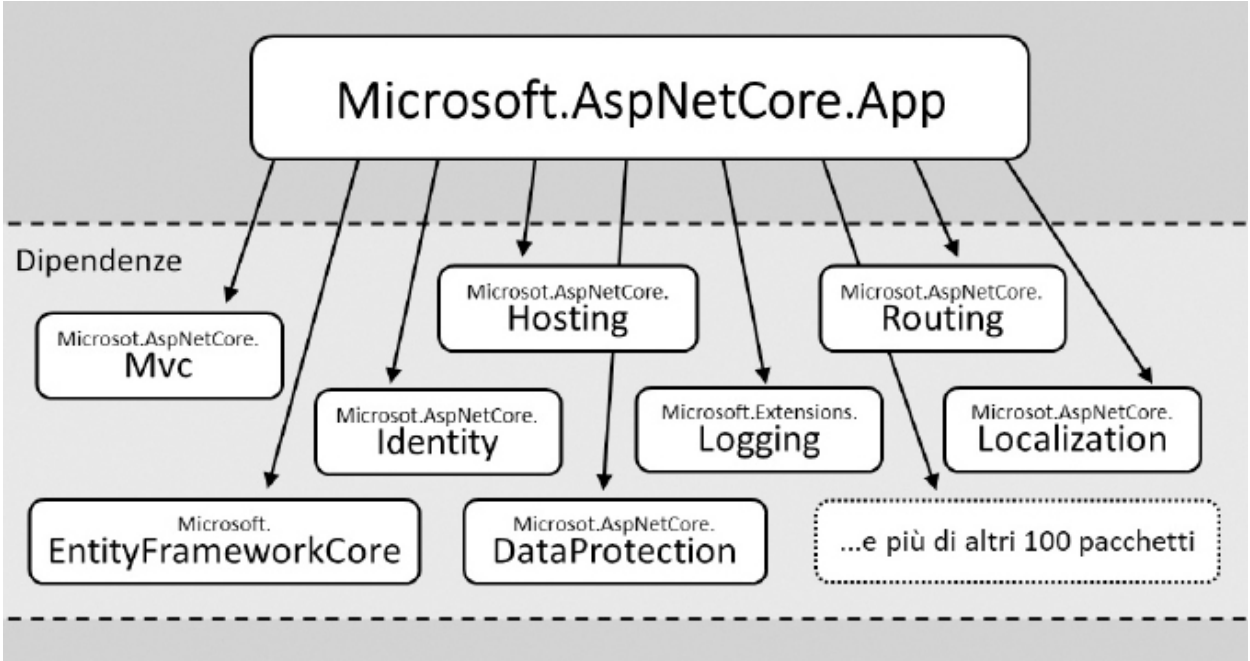


Figura 3.10 – Alcune delle dipendenze del metapacchetto NuGet Microsoft.AspNetCore.App.

A partire da ASP.NET Core 2.1, il metapacchetto `Microsoft.AspNetCore.App` referenzia solamente tecnologie su cui il team di sviluppo di ASP.NET Core ha il completo controllo. Perciò, se intendiamo sviluppare la nostra applicazione usando tecnologie di terze parti come Sqlite o Redis, dovremo aggiungere manualmente il riferimento a pacchetti come `Microsoft.Data.Sqlite` o `Microsoft.Extensions.Caching.Redis`. In ASP.NET Core 2.0, invece, tali tecnologie erano referenziate dal metapacchetto `Microsoft.AspNetCore.All`, il cui uso è ora sconsigliato.

Il tool Microsoft.VisualStudio.Web.CodeGeneration.Tools

Creare il primo progetto ASP.NET Core è solo l'inizio di un viaggio: la vera sfida consiste nel realizzare un'applicazione interessante per i nostri utenti.

Per facilitare almeno i compiti più ripetitivi, il progetto referenzia il tool `Microsoft.VisualStudio.Web.CodeGeneration.Tools` che aggiunge un ulteriore sottocomando `aspnet-codegenerator` al .NET Core CLI. Per ottenere le istruzioni sul suo utilizzo, digitiamo quanto segue da riga di comando.

Esempio 3.21

```
dotnet aspnet-codegenerator
```

Il tool è un generatore di codice per lo scaffolding di classi e UI. Permette cioè di aggiungere nuovi file al progetto contenenti una certa quantità di codice di partenza, allo scopo di velocizzare lo sviluppo. Il tool fornisce soltanto il motore di generazione, mentre gli effettivi template di scaffolding andranno aggiunti come pacchetti NuGet.

Il pacchetto `Microsoft.VisualStudio.Web.CodeGeneration.Design`, per esempio, fornisce i template per lo scaffolding di Controller, Aree, View e Razor Pages per ASP.NET MVC.

Conclusioni

Nel corso di questo capitolo abbiamo visto come ASP.NET Core possa essere configurato fin nei minimi dettagli. La sua architettura è abbastanza estendibile da permettere la sostituzione di interi componenti, così che possiamo scegliere il giusto rapporto tra funzionalità e prestazioni per ogni nostra applicazione. Nonostante il grande numero di possibilità, Microsoft ha deciso di ridurre al minimo la configurazione necessaria per iniziare a usare ASP.NET Core con le sue impostazioni di default. In questo modo, chi si avvicina al framework per la prima volta non ne uscirà disorientato. Questo è il giusto connubio che accontenta, senza compromessi, sia chi sviluppa per hobby sia i professionisti che intendono ottenere il massimo dalla propria applicazione.

Comprendere la struttura di un progetto ASP.NET Core è stato un passo essenziale, soprattutto per affrontare i prossimi capitoli, in cui muoveremo i primi passi con ASP.NET Core MVC, un modello di programmazione per siti web e web API basato sul pattern Model-View-Controller, che promuove la separazione delle responsabilità.

4
Primi passi con ASP.NET Core MVC

Se siamo sviluppatori ASP.NET di lungo corso, saremo già abituati ad ASP.NET Web Forms. Tradizionalmente, ASP.NET Web Forms, ha fornito un eccellente livello di astrazione nell'ambito della realizzazione di pagine web, volto a colmare tutti quei limiti che contraddistinguono il protocollo HTTP. Il modello di sviluppo proposto, infatti, è per molti aspetti analogo a quello delle applicazioni client, basato su oggetti che reagiscono alle azioni dell'utente sollevando eventi, e la stessa natura stateless del web diventa assolutamente impercettibile, grazie al ViewState che è in grado di mantenere lo stato del succedersi delle richieste.

Si tratta di un modello di sviluppo che, negli anni, si è dimostrato assolutamente valido per realizzare applicazioni di qualsiasi livello di complessità, ma che di certo non è esente da difetti: il runtime di ASP.NET, l'infrastruttura delle pagine e dei controlli server, infatti, è un insieme monolitico, difficilmente scindibile nelle sue singole componenti e quindi, per esempio, difficilmente testabile tramite unit test, o in cui, come sviluppatori, abbiamo un controllo limitato sull'effettivo markup generato per le singole pagine. Questi limiti e le pressanti richieste da parte della community di sviluppatori hanno portato Microsoft a pensare a una nuova piattaforma per lo sviluppo su web, alternativa ad ASP.NET Web Forms: stiamo parlando di ASP.NET MVC.

Con ASP.NET Core, Microsoft ha deciso che ASP.NET MVC fosse l'unico tra i due in grado di garantire un futuro alla propria piattaforma applicativa: quindi, all'interno di ASP.NET Core troviamo il supporto per una particolare versione di ASP.NET MVC, chiamata ASP.NET Core MVC (o, spesso, ASP.NET MVC Core), che è stata creata a partire da quanto ASP.NET MVC ha introdotto, ma con numerose e interessanti migliorie che andremo a scoprire nel corso dei prossimi capitoli.

Giunto alla versione 2, ASP.NET Core MVC è un framework decisamente maturo e pronto a essere utilizzato per applicazioni anche complesse, che propone un modello di sviluppo moderno, più aderente al funzionamento del web e basato sul pattern Model-View-Controller, uno dei più noti e collaudati pattern per l'architettura del layer di presentazione.

All'implementazione di MVC all'interno di ASP.NET Core è dedicata buona parte di questo libro, poiché, a meno che non sviluppiamo servizi, è molto probabile che avremo a che fare con questo pezzo dell'architettura di ASP.NET Core. Nel corso di questo capitolo e dei prossimi, cercheremo di mettere in luce le peculiarità di questa implementazione e di illustrare come sfruttarne a fondo l'estrema versatilità. In questo capitolo, in particolare, ne daremo una panoramica introduttiva in modo che possiamo, sin da subito, iniziare a familiarizzare con questo modello. Ne consigliamo una rapida lettura anche a chi ha già una buona dimestichezza con ASP.NET MVC.

Il pattern Model-View-Controller

Tutte le volte che dobbiamo gestire un elevato grado di complessità, la soluzione più adeguata è quella di disegnare un'architettura basata su oggetti più semplici, suddividendo tra essi le diverse responsabilità. Questo concetto si applica anche all'interfaccia utente, tant'è che, in letteratura, sono stati formalizzati diversi pattern secondo cui strutturare questo particolare strato applicativo. Tra essi, uno dei più diffusi e collaudati, negli ultimi anni, è il Model-View-Controller (MVC). Formulato per la prima volta intorno alla fine degli anni '70, è oggi alla base di numerose tecnologie di sviluppo per il web, tra cui ricordiamo Java Server Pages, Ruby on Rails e, ovviamente, ASP.NET MVC. Lo schema concettuale è rappresentato nella Figura 4.1.

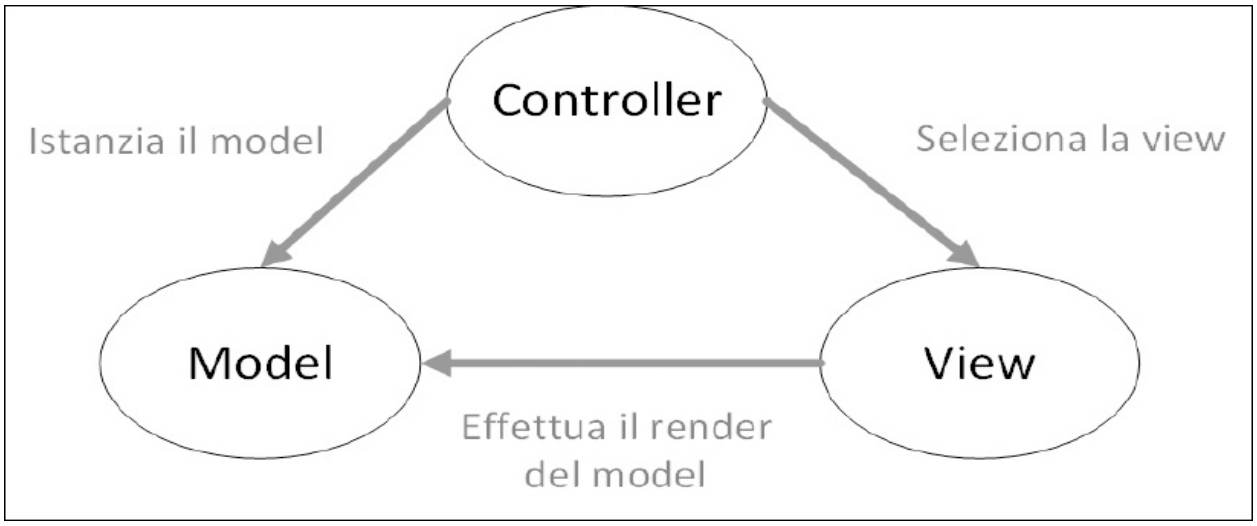


Figura 4.1 – Schema concettuale del pattern Model-View-Controller.

In esso distinguiamo i tre componenti che seguono.

- Il Model rappresenta il dato che deve essere mostrato sull'interfaccia; svolge pertanto il ruolo di contenitore di informazioni, ma implementa anche le logiche per dialogare con lo strato di business dell'applicazione stessa;

- Il Controller ha il compito di interpretare la richiesta dell'utente, di instanziare e valorizzare il model opportuno e, successivamente, di instradare la richiesta verso la view;

- La View è invece il template secondo cui il model deve essere rappresentato. Questo componente, pertanto, non contiene alcuna logica applicativa, ma solo l'eventuale logica necessaria per visualizzare correttamente il dato fornito come input.

In ASP.NET Core MVC, pertanto, il flusso di richiesta di una pagina e la generazione del markup di risposta non avviene in base alla composizione di controlli server side, come nel caso di ASP.NET Web Forms, ma tramite queste tre componenti del pattern Model-View-Controller. Per capire le modalità secondo cui questi oggetti interagiscono, il modo migliore è quello di analizzare un progetto di esempio, che sarà argomento della prossima parte di questo capitolo.

I servizi di routing

Quando l'applicazione riceve una richiesta da parte di un browser, il runtime di ASP.NET Core, attraverso i suoi middleware, presentati nei capitoli precedenti, deve determinare un oggetto in grado di soddisfarla e di produrre il relativo markup HTML di risposta. Nel caso di ASP.NET Core MVC, questo oggetto prende il nome di controller. Come impareremo nelle prossime pagine, un controller possiede la caratteristica di esporre una serie di gestori per le differenti richieste che pervengono all'applicazione, denominate action.

Ovviamente, visto che le richieste all'applicazione avvengono sotto forma di un URL, è necessario specificare da qualche parte quali sono le corrispondenze tra essi e i controller/action necessari per soddisfarle. Per questa necessità ASP.NET MVC sfrutta l'infrastruttura di routing, che altro non è che un sistema attraverso il quale possiamo definire dei percorsi HTTP e come una chiamata a ciascuno di essi debba comportarsi.

Generalmente, questo avviene mediante l'utilizzo di un opportuno middleware, già introdotto nel Capitolo 3, e che qui riportiamo per riprendere il discorso nell'Esempio 4.1. Il contenuto di questo codice è all'interno del metodo `Configure` all'interno del file `Startup.cs`.

Esempio 4.1

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
      …
      app.UseMvc(routes =>
      {
                routes.MapRoute(
                          name: "default",
                          template: "{controller=Home}/{action=Index}/{id?}");
      });
}
```

Il mapping che per default viene inserito in ogni applicazione ASP.NET Core MVC, serve a gestire la mancanza della tipica corrispondenza tra un URL e un file fisico sul server, sostituita invece dall'individuazione del controller e della relativa action in base al valore delle due componenti sull'indirizzo.

> *La sintassi utilizzata per definire la regola di route sfrutta l'extension method* `MapRoute` *che, internamente, implementa 'interfaccia* `IRouteBuilder`*, alla base del routing di ASP.NET Core MVC.*

Grazie a questo middleware, quindi, una richiesta all'indirizzo */People/Index* verrà instradata presso un controller denominato `People`, e in particolare verso la sua action `Index`; analogamente, */Countries/Edit/1* corrisponderà alla action `Edit` del controller Countries, fornendo come parametro id il valore 1. Il metodo `MapRoute` ci permette anche di definire dei valori di default, nel caso qualcuno di questi parametri dovesse mancare. In virtù di questi parametri, alla root del sito `http://localhost/` corrisponderà il controller *Home* e la sua action *Index*. Ovviamente queste impostazioni possono essere personalizzate secondo la nostra necessità, sia modificando la route di default sia aggiungendo delle route ulteriori.

Ora che abbiamo compreso il meccanismo secondo il quale viene determinato il controller a cui demandare l'onere di soddisfare una richiesta, possiamo spostare la nostra attenzione su come effettivamente realizzarne uno.

Il controller e il model

Dal punto di vista strettamente pratico, un controller non è altro che una classe che implementa l'interfaccia `IController`. Sebbene sia quindi sufficiente creare una nuova classe all'interno del progetto, in Visual Studio possiamo anche usare la funzionalità Add Controller, presente nel menu contestuale di un progetto ASP.NET MVC, che apre la finestra di dialogo mostrata nella Figura 4.2.

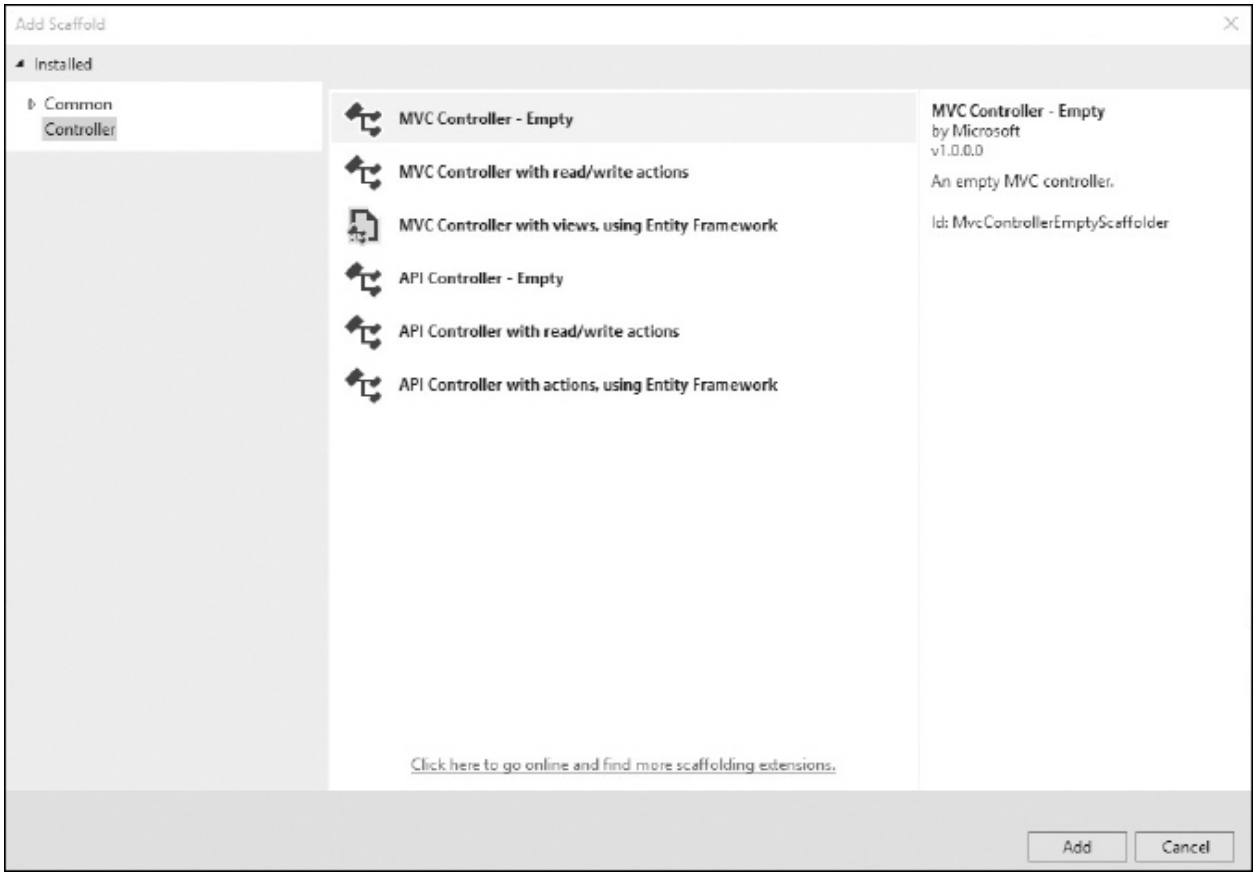


Figura 4.2 – Finestra di dialogo per la creazione di un nuovo controller.

Tramite questa maschera possiamo specificare il nome, il quale per convenzione viene fatto terminare con il suffisso *-Controller*, e il template che vogliamo utilizzare per generarlo. Se utilizziamo le impostazioni visibili nella figura, verrà aggiunta al progetto una nuova classe `HomeController`, il cui codice è quello dell'Esempio 4.2.

Esempio 4.2

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      return View();
   }
}
```

Come possiamo notare, `HomeController`, in realtà, eredita dalla classe base `Controller`, che internamente implementa già tutti i membri richiesti dall'interfaccia `IController`, e contiene la definizione della action `Index`, che non è altro che un metodo pubblico definito al suo interno; esso non presenta alcun tipo di logica e si limita a restituire come risultato una view e, in particolare, visto che non abbiamo specificato alcuna informazione addizionale, la view predefinita (individuata attraverso un meccanismo di convenzioni, che vedremo successivamente). Tipicamente, all'interno di una action abbiamo anche il compito di recuperare e valorizzare i dati che vogliamo rappresentare tramite la view. Come abbiamo accennato, il contenitore di questi dati è il model. Per casi semplici come quello che stiamo esaminando, la classe base `Controller` espone la proprietà `ViewBag`, ossia un oggetto di tipo `dynamic`, che è accessibile anche dalla view e che possiamo valorizzare con qualsiasi tipo di informazione, come nell'Esempio 4.3.

Esempio 4.3

```
public ActionResult Index()
{
   ViewBag.Message = "Ciao da ASP.NET Core";
   ViewBag.CurrentDate = DateTime.Now.ToString();
   ViewBag.ShowDate = true;
   return View();
}
```

All'interno del controller troverà spazio tutta la logica necessaria a recuperare i dati, manipolarli e poi rimandarli indietro al client, secondo la logica che ci siamo prefissati.

In realtà, in ASP.NET Core MVC non è obbligatorio, per un controller, né ereditare da una classe che implementi l'interfaccia `IController`, né restituire un tipo `IActionResult` come risultato di una action. In effetti, ASP.NET Core MVC supporta pienamente il concetto di controller POCO (Plain Old CLR Object - i vecchi cari oggetti del CLR). In estrema sintesi, un controller che non debba utilizzare i servizi di ASP.NET Core MVC può fare a meno di una classe base e può restituire qualsiasi tipo serializzabile.

Esempio 4.4

```
public class PocoController
{
   public Date CurrentTime()
   {
      return DateTime.Now;
   }
}
```

L'output della action dell'Esempio 4.4 sarà la serializzazione del tipo di ritorno – in questo caso una data – nel formato JSON. Diventa così molto più semplice, in estrema analisi, tenere action con scopi differenti all'interno dello stesso controller, semplificando la logica in quegli scenari in cui si fa ampio uso di endpoint che restituiscono JSON, per esempio in applicazioni SPA o AJAX. Avremo modo di tornare ampiamente su questi temi nel prossimo capitolo.

Giunti a questo punto, manca solo l'ultimo tassello per completare la nostra prima pagina: dobbiamo creare la nostra prima view.

La view e gli HTML helper

Come abbiamo più volte avuto modo di sottolineare durante il capitolo, la view è il componente del pattern MVC responsabile di rappresentare in forma di markup, visto che siamo nell'ambito di un'applicazione web, i dati contenuti all'interno del model.

In un progetto ASP.NET Core MVC, esse sono memorizzate all'interno della directory `Views`, disposta nella struttura di directory che vediamo nella Figura 4.3.

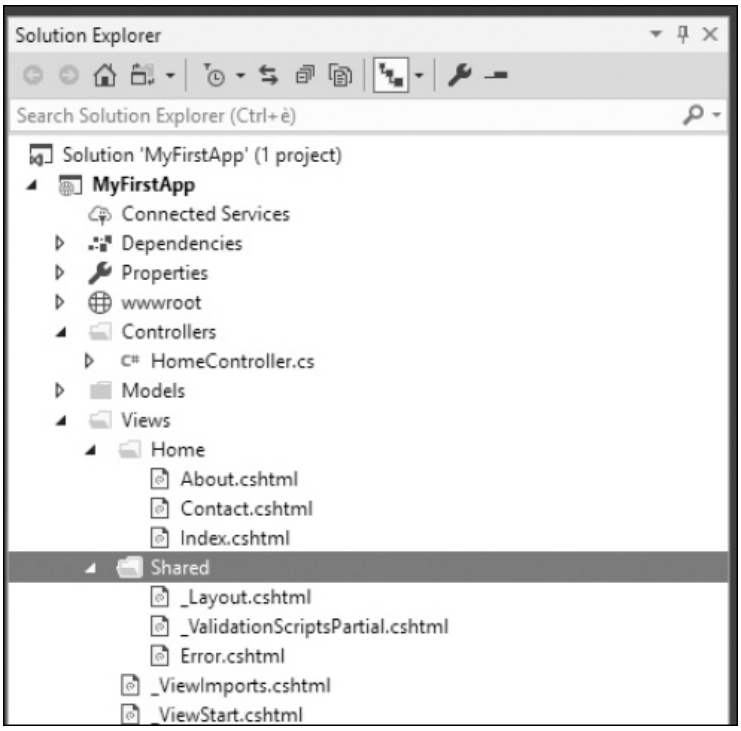


Figura 4.3 – Struttura di directory per le view.

La directory `Views`, a propria volta, contiene una sottodirectory per ogni controller. All'interno di queste ultime trovano posto i file, con estensione `.cshtml`: si tratta di file che vengono processati da un engine, denominato Razor, che, a seguito di un'operazione di parsing, produrrà delle classi (in C#) che rappresentano la struttura del nostro markup e che saranno poi eseguite effettivamente dal runtime.

Se utilizziamo Visual Studio, non dobbiamo preoccuparci di creare a mano la directory di un controller perché, anche in questo caso, Visual Studio ci mette a disposizione una comoda finestra di dialogo, mostrata nella Figura 4.4, per generare una view. Per aprirla non dobbiamo far altro che selezionare l'opzione *Add View* dal menu contestuale che si ottiene dopo un click con il tasto destro sul codice della action.

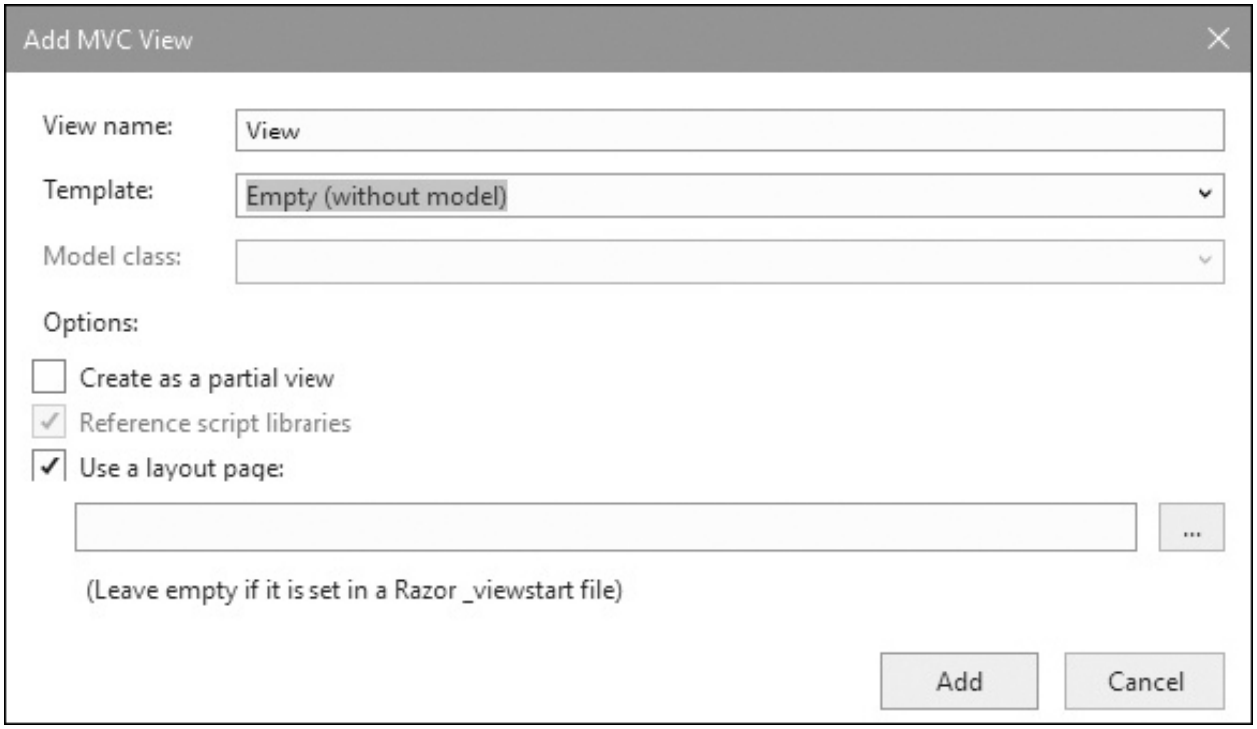


Figura 4.4 – Finestra di dialogo per la creazione di una nuova view.

Utilizzando un qualsiasi altro editor (come, per esempio, Visual Studio Code) l'operazione è fattibile manualmente, senza troppi patemi.

Come possiamo notare, il nome che ci viene proposto da Visual Studio coincide con quello della action da cui siamo partiti. In questo caso, prende il nome di view predefinita per quella determinata action, e potrà essere referenziata tramite il metodo `View`, come abbiamo visto nell'Esempio 4.4, senza specificare esplicitamente il nome. La dialog mostra anche ulteriori opzioni, di cui parleremo in maniera approfondita nel corso del Capitolo 6 e nei successivi. Per il momento, quindi, tralasciamole e proseguiamo effettuando un click sul pulsante *Add*; il risultato è la creazione di un nuovo file, il cui contenuto è mostrato nell'Esempio 4.5.

Esempio 4.5

```
@{
   ViewBag.Title = "Index";
}

<h2>Index</h2>
```

Come possiamo notare, si tratta di una sintassi davvero particolare, che concilia all'interno dello stesso file la presenza di markup HTML con il codice C#. A prescindere da quale linguaggio stiamo effettivamente utilizzando, nel codice precedente possiamo distinguere fondamentalmente due "zone":

- Un blocco di codice, delimitato da `@{ ... }` in C#, all'interno del quale accediamo al contenuto della variabile `ViewBag`, che abbiamo già avuto modo di conoscere nella sezione precedente, valorizzandone la proprietà `Title`.
- Una sezione di markup HTML, costituita da un tag H2.

Nel codice di una view, grazie al carattere `@`, possiamo cambiare il contesto da codice a markup, e viceversa.

Esempio 4.6

```
<h2>Index</h2>

<p>@ViewBag.Message</p>

@if (ViewBag.ShowDate)
{
   <div>La data corrente è @ViewBag.CurrentDate</div>
}

@Html.ActionLink("Un link...", "SayHello")
```

Il codice dell'Esempio 4.6 mostra alcune peculiarità della sintassi di Razor.

- Come abbiamo accennato, una volta inserito il tag `<p>` (contesto markup), possiamo passare al contesto codice tramite il carattere `@` e referenziare il contenuto di `ViewBag`, che abbiamo valorizzato nel controller;
- nel realizzare il template, potremmo aver bisogno dei tipici costrutti di branching o di iterazione e, per usufruirne, non dobbiamo far altro che passare al contesto del codice. Per esempio nel codice precedente abbiamo utilizzato un blocco `if` per visualizzare o meno la data corrente al variare del parametro `ShowDate`;
- l'istruzione `@Html.ActionLink` appartiene a una categoria di metodi parecchio utilizzati nella realizzazione di pagine con Razor, e che sono denominati HTML helper. Nello specifico, `ActionLink` serve a generare un link verso la action "SayHello" dello stesso controller, con il testo specificato; sebbene possiamo comunque inserire un link sfruttando il tag `<a>`, questo metodo è preferibile perché l'URL generato dipende dalle impostazioni del routing e, nel caso queste vengano cambiate in futuro, anche il link verrà modificato automaticamente di conseguenza;
- l'engine è sufficientemente evoluto da riuscire a discernere i casi in cui l'utilizzo del carattere `@` non implica un cambio di contesto.

Il risultato di questa nostra prima pagina realizzata con ASP.NET Core MVC è visibile nella Figura 4.5.

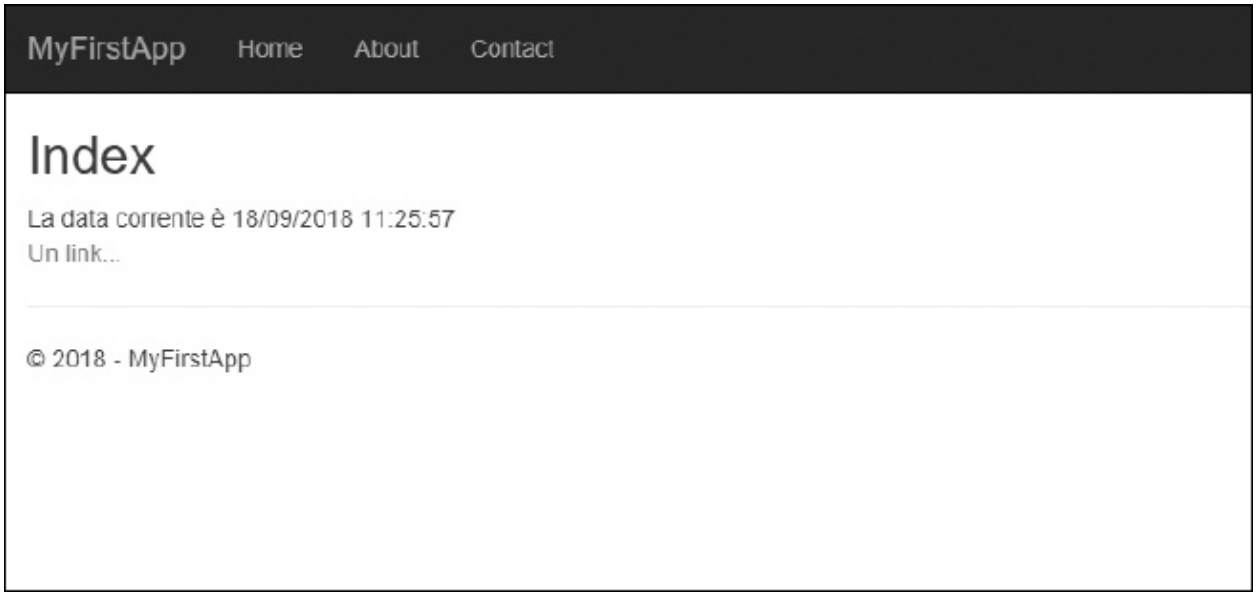


Figura 4.5 – La nostra prima pagina in ASP.NET Core MVC.

Se proviamo a dare un'occhiata al sorgente della pagina, mostrato nell'Esempio 4.6, possiamo effettivamente toccare con mano la sostanziale differenza rispetto ad altri framework più complessi: il contenuto della pagina è molto semplice, non sono presenti view state o control state, e il markup generato è assolutamente sovrapponibile a quanto abbiamo scritto nella view.

Esempio 4.7 - HTML

```
<!DOCTYPE html>
<html>
   <head>
      <!-- altro markup qui -->
   </head>
   <body>
      <h2>Index</h2>
      <p>Ciao da ASP.NET MVC</p>
      <div>La data corrente è 15/06/2018 11:48:11</div>
      <a href="/Home/SayHello">Un link...</a>
      <script src="/Scripts/jquery.js"></script>
   </body>
</html>
```

Come possiamo notare, esistono comunque degli elementi addizionali, estranei alla view che abbiamo realizzato, come la sezione `<head>` della pagina o il riferimento alla libreria jQuery. Essi sono stati inseriti da quella che in ASP.NET Core MVC è chiamata layout page

e che si trova all'interno della directory `Views\Shared`, è chiamata `_Layout.cshtml` e il cui contenuto tipo è quello dell'Esempio 4.8.

Esempio 4.8

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <meta name="viewport" content="width=device-width" />
   <title>@ViewBag.Title</title>
   @Styles.Render("~/Content/css") </head>
<body>
   @RenderBody()
   @Scripts.Render("~/bundles/jquery")
   @RenderSection("scripts", required: false)
</body>
</html>
```

In questa fase non è necessario entrare nel dettaglio di ogni singola riga di codice di questa view, di cui parleremo approfonditamente nel corso dei prossimi capitoli. L'unico aspetto da notare, al momento, è la presenza del metodo `RenderBody`, in corrispondenza del quale verrà inserito il markup prodotto dalla specifica view.

Nonostante si sia trattato di un esempio assolutamente banale, ci è comunque servito ad apprendere molto del funzionamento e delle logiche che regolano il modello di ASP.NET Core MVC: abbiamo visto come, alla ricezione di una richiesta, il primo passaggio eseguito dal runtime sia l'instradamento tramite routing verso un controller e, più in dettaglio verso una action; quest'ultima, come risultato, ha restituito una view, ossia un template che abbiamo costruito integrando markup e codice, tramite la particolare sintassi dell'engine Razor.

Al momento, però, non siamo ancora in grado di gestire l'input dell'utente, per esempio un form che possa effettuare un POST verso il nostro sito web. Nella prossima sezione proveremo a colmare questa lacuna.

Gestire una form di input

Il funzionamento di ASP.NET Core MVC, di fronte all'input che arriva da una form, è ancora una volta orientato alla semplicità: è privo di astrazioni e maggiormente aderente al funzionamento del protocollo HTTP in generale. Una richiesta che proviene da una form viene gestita in maniera analoga a tutte le altre, individuando la coppia controller/action corrispondente e processandola.

Cerchiamo di chiarire meglio il concetto con un esempio. Per poter consentire all'utente di inserire dei dati, intanto dobbiamo predisporre una pagina apposita, e quindi una action come la seguente.

Esempio 4.9

```
public ActionResult SayHello()
{
   return this.View();
}
```

Il codice dell'Esempio 4.9 è assolutamente banale e non merita alcun commento; ben più interessante è invece il contenuto della view, mostrato nell'Esempio 4.10.

Esempio 4.10

```
@using (Html.BeginForm())
{
   <div>
      <p>Inserisci il tuo nome: </p>
      <p>@Html.TextBox("name")</p>
      <input type="submit" value="Go!" />
   </div>
}

<div>@ViewBag.Message</div>
```

Nel codice in alto abbiamo utilizzato un paio di HTML helper che, nello sviluppo di applicazioni ASP.NET Core MVC, si rivelano sicuramente utili. Il primo di questi helper è `BeginForm`, che serve a produrre il tag `<form>`; la modalità di utilizzo è leggermente diversa a quella di `ActionLink`, che abbiamo visto in precedenza, visto che va inserito all'interno di un blocco `using`, così che abbiamo la possibilità di specificare con esattezza anche il punto in cui termina. Successivamente, abbiamo sfruttato l'HTML helper `TextBox`, per generare un tag `<input type="text" />` all'interno del quale l'utente potrà inserire il suo nome.

Al click sul bottone di submit, il browser dell'utente invierà in POST il contenuto della form al medesimo indirizzo della richiesta precedente, ossia */Home/SayHello*. Dal nostro punto di vista, questo si tradurrà nell'esecuzione di un'ulteriore action che, in base alle regole di routing, dovrà comunque chiamarsi `SayHello`, ma sarà specifica per gestire la chiamata in POST.

Esempio 4.11

```
[HttpPost]
public ActionResult SayHello(string name)
{
   ViewBag.Message = string.Format(""Ciao {0}!", name);
   return this.View();
}
```

Anche in questo caso, ci sono alcuni aspetti da sottolineare: questa action, seppure omonima a quella che abbiamo realizzato in precedenza, è marcata con l'attributo `HttpPost`, in modo da segnalare che dovrà essere utilizzata per gestire le richieste di tipo POST. Inoltre, come possiamo notare, essa presenta un parametro `name`, che poi abbiamo utilizzato per produrre il messaggio di risposta. Come vedremo nel corso del Capitolo 15, è compito del runtime di ASP.NET Core MVC, e in particolare di un oggetto denominato model binder, quello di valorizzare questi parametri in base al contenuto della richiesta. Per ora, ci basta sapere che, in virtù del fatto che il suo nome corrisponde a quello che abbiamo utilizzato nell'html helper `TextBox`, esso conterrà effettivamente il dato inserito dall'utente nella form.

Dopo aver valorizzato opportunamente la `ViewBag`, l'ultima riga di codice dell'Esempio 4.11 restituisce la stessa view `SayHello.cshtml` dell'Esempio 4.10, così che il messaggio possa effettivamente essere mostrato.

Ancora una volta, si è trattato di un esempio piuttosto semplice (il classico "hello world"), ma che ci aiuta a capire i meccanismi basilari di ASP.NET Core MVC anche nel caso della gestione dell'input utente. Sotto questo punto di vista, ovviamente, c'è ancora tanto da dire, a partire dalle modalità per realizzare form complesse fino ad arrivare a spiegare come impostare regole di validazione. Ci occuperemo in maniera approfondita di questo argomento nel corso del Capitolo 15.

Per il momento, invece, cambiamo momentaneamente argomento e torniamo a parlare della struttura di un progetto ASP.NET Core MVC e di come possiamo organizzare le varie componenti di un progetto, nel momento in cui la complessità dell'applicazione e il numero di pagine aumentano.

ASP.NET Core MVC e progetti complessi: le aree

Come abbiamo avuto modo di vedere nel corso di questo capitolo, in ASP.NET Core MVC la struttura di directory del progetto non rispecchia l'effettivo path delle pagine, che invece è determinato esclusivamente dalle regole di routing.

Al contrario, esse hanno esclusivamente la funzione di contenitori, secondo una struttura ben definita che stabilisce dove posizionare i vari file di model, controller e view.

A rigor del vero, bisogna comunque specificare che il framework impone che le convenzioni sulle directory siano rispettate solo per quanto riguarda le view, mentre non è necessario che model e controller si trovino nelle directory omonime. I concetti espressi da questa sezione rimangono comunque validi.

Questa impostazione, però, può costituire un problema quando le dimensioni del progetto aumentano, a causa del proliferare di file, o anche quando si vuole suddividere le varie funzionalità in "aree tematiche". Pensiamo, per esempio, al caso di un CMS, in cui magari abbiamo sia una parte pubblica, visibile a tutti, sia una sezione di backoffice amministrativo, anche con differenti layout, pattern di routing e regole di accesso: si tratta delle tipiche necessità per le quali, in ASP.NET MVC, è stato introdotto il concetto di Area, che troviamo anche all'interno di ASP.NET Core MVC. Da Visual Studio, se effettuiamo un click con il tasto destro del mouse sul progetto, dal menu contestuale possiamo aggiungere una nuova area tramite il comando *Add Area* che vediamo nella Figura 4.6.

A questo punto, se diamo un nome alla nuova area, per esempio Backoffice, e confermiamo, viene creata nel progetto una nuova struttura di directory, del tutto simile alla principale, come mostrato nella Figura 4.7.

Non utilizzando Visual Studio, l'operazione si può portare a termine facilmente, creando manualmente una struttura come quella riportata nell'immagine.

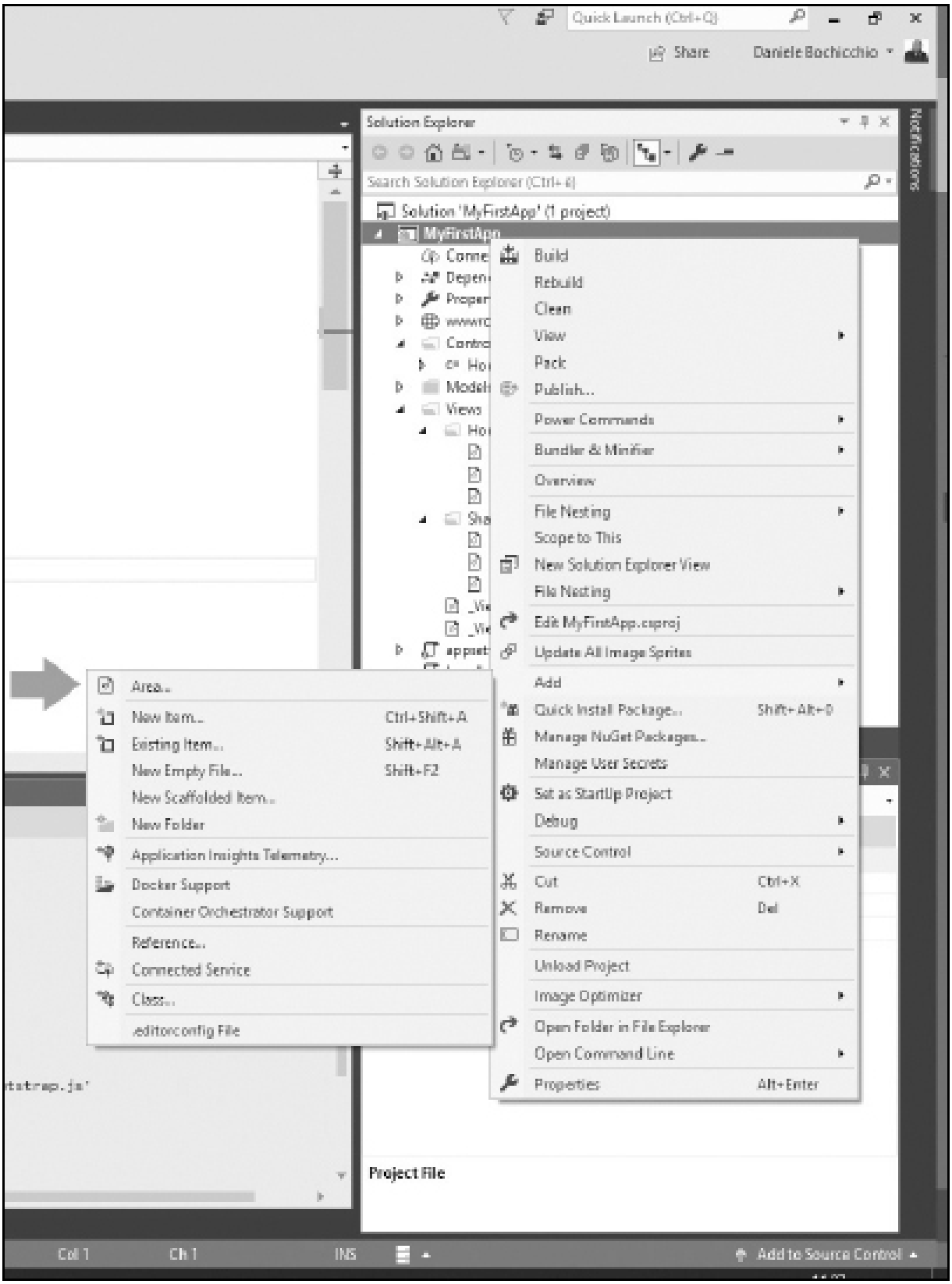


Figura 4.6 – Menu contestuale per l'aggiunta di un'area.

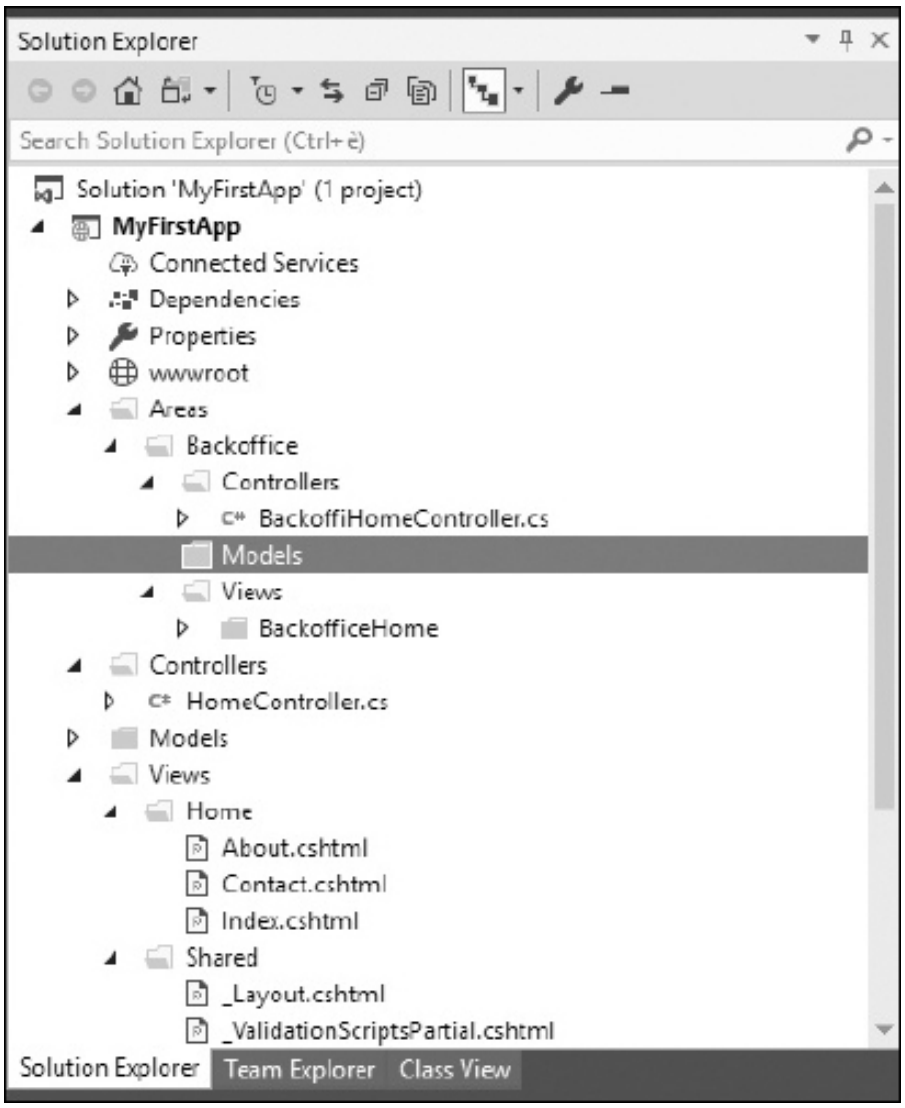


Figura 4.7 – La struttura delle directory dell'area Backoffice.

In questo modo abbiamo definito una sezione, separata dal resto del progetto, all'interno della quale implementare la nuova area funzionale dell'applicazione: abbiamo una directory specifica per controller e model, oltre alla struttura relativa alle view. Questo ci consente, per esempio, di definire una layout page specifica, che sarà utilizzata solo dalle pagine dell'area backoffice.

Per completare l'operazione e fare in modo che tutto funzioni, dobbiamo leggermente cambiare la parte relativa al routing, per supportare una nuova convenzione, basata sull'aggiunta di un'area.

Esempio 4.12

```
app.UseMvc(routes =>
{
   route.MapRoute(
      name : "areas",
      template : "{area:exists}/{controller=Home}/{action=Index}/{id?}"
   );
   …
   // mantenere il routing già presente
});
```

La registrazione del routing per le aree va fatta una sola volta e deve precedere quella più generale, che abbiamo visto prima, perché le stesse agiscono secondo un algoritmo di corto circuito: la prima vera fa saltare la valutazione delle altre regole, quindi quella più generica va tenuta sempre per ultima. La ricerca di una view verrà effettuata secondo questa logica:

- `/Areas/<Area-Name>/Views/<Controller-Name>/<Action-Name>.cshtml`
- `/Areas/<Area-Name>/Views/Shared/<Action-Name>.cshtml`
- `/Views/Shared/<Action-Name>.cshtml`

Inoltre, a differenza di ASP.NET MVC, in ASP.NET Core MVC è necessario decorare ciascun controller con l'attributo `Area`, come nell'esempio che segue.

Esempio 4.13

```
[Area("Backoffice")]
public class AdminController : Controller
{
   public IActionResult Index()
   {
      return View();
   }
}
```

Nel corso di questo capitolo abbiamo accennato all'HTML helper `ActionLink`, tramite cui possiamo generare dei link verso altre pagine dell'applicazione, basandoci sul nome del controller e della action piuttosto che sull'indirizzo fisico. In generale, questo metodo punta sempre all'area della pagina corrente. Non esiste uno specifico overload che ci consenta di specificare il nome dell'area, pertanto se vogliamo creare un link dall'area principale all'area di backoffice, dobbiamo utilizzare la tecnica dell'Esempio 4.14.

Esempio 4.14

```
@Html.ActionLink("Vai al backoffice", "Index", "Admin",
new { Area = "Backoffice" }, null)
```

Nel codice precedente abbiamo sfruttato un anonymous type per specificare un parametro addizionale del routing, ossia il nome dell'area. Il risultato in pagina, pertanto, sarà un link alla action `Index` di `AdminController`, contenuto nell'area denominata Backoffice.

Conclusioni

In questo capitolo abbiamo voluto introdurre i concetti basilari di ASP.NET Core MVC, che poi verranno esplorati in maggiore dettaglio nel prosieguo del libro. ASP.NET Core MVC è il framework per lo sviluppo di applicazioni web, basato sul popolare pattern Model-View-Controller, che impareremo a utilizzare con ASP.NET Core. Nei fatti, quando parliamo di ASP.NET Core, al 90% ci riferiamo anche ad ASP.NET Core MVC, che rappresenta il servizio certamente più utilizzato fra quelli messi a disposizione.

Dopo aver visto le componenti che lo contraddistinguono e le responsabilità di ognuna, abbiamo creato il nostro primo progetto in Visual Studio, sfruttando uno dei molteplici template presenti, per poi dedicarci alla creazione della nostra prima pagina. Si è trattato di un esempio molto semplice, che però ci ha fatto apprezzare la semplicità del modello di sviluppo basato sul pattern MVC e il totale controllo che abbiamo sul markup effettivamente prodotto in pagina. Anche per quanto riguarda la gestione delle form di input, sebbene abbiamo solo iniziato a illustrare la problematica, che sarà affrontata in maniera approfondita nel capitolo successivo, possiamo già renderci conto di come la filosofia di base non cambi, dovendo comunque esporre una action a cui è demandata la gestione della request di POST.

Come ultimo argomento del capitolo abbiamo introdotto il concetto di area, tramite la quale possiamo partizionare e organizzare i file di progetto in diverse aree funzionali, ognuna caratterizzata dai propri controller, view e regole di routing.

A questo punto abbiamo tutte le nozioni di base necessarie per affrontare maggiormente nel dettaglio le varie caratteristiche di ASP.NET Core MVC, a partire dal prossimo capitolo, in cui ci occuperemo dei controller.

5
Controller e routing

Nel capitolo precedente abbiamo introdotto il modello di sviluppo proposto da ASP.NET Core MVC, realizzando due pagine di esempio che, per quanto semplici, ci hanno aiutato a comprendere i meccanismi che ne regolano il funzionamento. Da questo capitolo e per i successivi, ci muoveremo invece più nel dettaglio, cercando di sviscerarne le singole componenti, partendo dai controller, proseguendo per le view, fino ad arrivare a trattare tematiche avanzate, utili per trarre il massimo vantaggio dalle nostre applicazioni.

In questo capitolo, in particolare, ci occuperemo dei controller, ossia dei principali responsabili nel flusso di gestione della request di ASP.NET Core MVC. Esploreremo nel dettaglio le modalità con cui possiamo definire controller e action, e le diverse tipologie di risposta che queste ultime possono fornire, sfruttando eventualmente anche le peculiarità dell'esecuzione asincrona, introdotta dal .NET Framework 4.5 e pienamente disponibile con .NET Core. Come ultimo argomento del capitolo, introdurremo poi il concetto dei filter, tramite i quali possiamo inserire della logica personalizzata nella pipeline di ASP.NET Core MVC, al fine di gestire problematiche quali logging, gestione degli errori o autorizzazioni, solo per citarne alcune.

Prima di inoltrarci in questi aspetti, però, è necessario fare un piccolo passo indietro, in merito al routing, che abbiamo introdotto nel capitolo precedente e che rappresenta il primo gestore che viene attivato, a seguito del quale deve essere avviata la procedura di instradamento verso la action designata.

URL Routing in ASP.NET Core MVC

Come abbiamo visto nel capitolo precedente, ASP.NET Core MVC non possiede il concetto di "pagina", inteso come un file fisico che possieda le informazioni necessarie per produrre una pagina HTML, ma basa il suo funzionamento sull'instradamento di una request verso il controller e la action, che dovranno, in prima istanza, gestirla. Tutto ciò è possibile grazie alla route di default, dichiarata tipicamente nel metodo `Configure` della classe `Startup`, che definisce la registrazione del middleware `RouteMiddleware` di ASP.NET Core MVC e che dichiara al suo interno il routing, la cui dichiarazione troviamo riportata nell'Esempio 5.1.

Esempio 5.1

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
      app.UseBrowserLink();
   }
   else
   {
      app.UseExceptionHandler("/Home/Error");
   }
   app.UseStaticFiles();
   app.UseMvc(routes =>
   {
      routes.MapRoute(
         name: "default",
         template: "{controller=Home}/{action=Index}/{id?}");
   });
}
```

Il codice in alto definisce una route i cui parametri sono rappresentati dal controller e dalla action da invocare, oltre a un eventuale identificativo, specificando anche i relativi valori di default. Scendendo maggiormente nel dettaglio, il primo aspetto che salta all'occhio è l'assenza dell'indicazione esplicita di un route handler per questa route; ciò è possibile perché abbiamo sfruttato l'extension method `MapRoute`, specifico di ASP.NET Core MVC, che internamente si occupa poi di referenziare l'oggetto `IRouteBuilder`.

Tecnicamente, le route vengono tutte valutate (poiché sono all'interno di una collection) dal `RouteMiddleware`, che chiama il metodo `RouteAsync` su ogni route registrata. Questo metodo ha la logica per capire se gestire la richiesta, impostando la proprietà `RouteContext.Handler` con un valore non nullo. Se una route imposta un handler per la richiesta, allora l'elaborazione continua con quest'ultima. In caso contrario, si procede con la valutazione delle altre route. Se nessuna route dovesse soddisfare la richiesta, il middleware del routing invoca il suo metodo `Next`, che segnala alla pipeline di esecuzione di ASP.NET Core che può invocare il middleware successivo.

Sia route sia middleware sono registrati nell'ordine di priorità (per primo quello che è meno generico), proprio per rendere possibile una maggiore flessibilità. Come abbiamo già anticipato nel capitolo precedente, diventa importante specificare il giusto ordine delle route e indicare per ultima quella più generica, che altrimenti avrà sempre il sopravvento sulle altre.

Constraint sulle route

Il metodo `MapRoute` ci consente anche di specificare constraint (cioè, dei vincoli da soddisfare) per la route, definiti tramite espressioni regolari o tramite classi che implementino l'interfaccia `IRouteConstraint`, tramite un overload del metodo `MapRoute`.

Se, per ipotesi, gli id della nostra applicazione sono tutti numerici, possiamo indicare questo vincolo con il codice dell'Esempio 5.2.

Il vantaggio di questo approccio è che, nel caso in cui venga fornito un parametro non valido, è la route stessa a non essere ritenuta valida; quando invece la richiesta è corretta, il runtime di ASP.NET Core MVC si occupa di eseguire il codice del controller selezionato.

In realtà, ASP.NET Core MVC supporta anche una variante di questo approccio, che consente di specificare la constraint direttamente nel parametro. L'esempio precedente diventa registrabile con una sintassi differente, mostrata nell'Esempio 5.3.

Esempio 5.2

```
routes.MapRoute(
   name: "numeric-id-route",
   template: "{controller=Home}/{action=Index}/{id?}",
   constraints: new { id = "\\d+" },
   defaults: null
);
```

Esempio 5.3

```
routes.MapRoute(
   name: "numeric-id-route",
template: "{controller=Home}/{action=Index}/{id:int}");
```

Si può notare l'utilizzo del parametro `{id:int}`, che consente di forzare il constraint direttamente in linea. I valori supportati sono:

- `int`: il valore passato deve essere un intero, anche con valori negativi;
- `bool`: il valore passato deve essere `true` o `false` (il case è irrilevante);
- `datetime`: il valore passato deve essere una data, in formato americano (o ISO);
- `decimal`: il valore passato deve essere un valore di tipo `decimal`, anche con valori negativi;
- `double`: il valore passato deve essere un valore di tipo `double`, anche con valori negativi;
- `float`: il valore passato deve essere un valore di tipo `float`, anche con valori negativi;
- `long`: il valore passato deve essere un valore di tipo `long`, anche con valori negativi;
- `guid`: corrisponde a un GUID;
- `minlength(value)` e `maxlength(value)`: il valore deve essere una stringa di almeno o al massimo il numero di caratteri specificati;
- `length(value)` e `length(min, max)`: consentono di stabilire una lunghezza esatta o minima e massima per una stringa;
- `min(value)` e `max(value)`: consentono di specificare un valore minimo o uno massimo;
- `range(min, max)`: consente di specificare un intervallo di validità del valore passato;
- `alpha`: forza il parametro a essere un carattere alfanumerico;
- `regex(expression)`: consente di specificare un'espressione di una regular expression, per indicare criteri di validazione anche complessi;
- `required`: indica che un campo è obbligatorio.

Per scenari più avanzati, poi, resta possibile implementare l'interfaccia `IRouteConstraint` e creare una classe che specifichi criteri più avanzati. In realtà, grazie all'ampia versatilità dei criteri appena elencati, questa necessità è confinata a scenari davvero particolari e raramente vi capiterà di doverne fare uso.

Ma a che cosa serve poter specificare così tante possibilità in fase di constraint di una route? Per esempio, potremmo decidere di avere una route che gestisce i profili utente, al cui interno andiamo a specificare dei parametri avanzati di controllo di validità. In questo caso andremo a variare leggermente la registrazione, poiché le constraint in linea non sono compatibili con la registrazione in maniera esplicita.

Esempio 5.4

```
routes.MapRoute(
   name: ”users-profiles”,
   template: ”users/{username:length(3,15)}”,
defaults: new {controller=”Users”, action=”Profile”});
```

Questa route soddisferà tutte le chiamate agli URL che iniziano con `Users` e che contengono un parametro, che sarà chiamato `username`, di almeno 3 caratteri e al massimo 15. Il motore invocherà la action `Profile` del controller `Users`, passando un parametro `username` di tipo stringa. In questo modo, il controller sarà in grado di restituire il risultato e, allo stesso tempo, noi saremo in grado di utilizzare URL la cui forma è decisamente più user friendly. Se nella nostra applicazione non esistono username più lunghi di 15 caratteri e più corti di 3, aggiungere una constraint del genere aiuta a migliorare la sicurezza, poiché diventa impossibile passare stringhe più complesse.

Abbiamo ampiamente parlato di controller sia nel capitolo precedente sia in questa prima parte. Ma cos’è effettivamente un controller? Nella sezione successiva cercheremo di dare una risposta dettagliata a questa domanda.

Anatomia di un controller

Nell’ambito del pattern MVC, il controller, come abbiamo avuto modo di ribadire più volte, è il principale responsabile della gestione di una richiesta proveniente dal browser. Esso ha il compito di istanziare il model, che conterrà i dati da utilizzare per popolare la risposta e selezionare la view più opportuna per rappresentarli. In ASP.NET Core MVC, come abbiamo visto nel capitolo precedente, questo ruolo può essere svolto da una qualsiasi classe che implementi `IController`: si tratta di un’interfaccia che espone il solo metodo `Execute` dell’Esempio 5.5.

Esempio 5.5

```
public interface IController
{
   void Execute(RequestContext requestContext);
}
```

Se volessimo implementare direttamente questa interfaccia, saremmo costretti a scrivere, all’interno del metodo `Execute`, tutta la logica per recuperare le informazioni dal routing e dalla `Request` corrente, e successivamente scrivere il risultato sulla `Response` dell’`HttpContext`. Nella realtà dei fatti, però, ciò non accade, perché ASP.NET Core MVC ci consente di lavorare a un livello di astrazione più elevato.

Quando creiamo un nuovo controller, infatti, tipicamente creiamo una classe che eredita dalla classe base `Controller`. Essa implementa ovviamente `IController`, ma espone una serie di funzionalità più evolute, di cui parleremo nelle prossime pagine.

Come abbiamo detto nel capitolo precedente, inoltre, ASP.NET Core MVC consente di utilizzare anche controller POCO (Plain Old CLR Object), senza che sia necessaria una classe base. Questo scenario si sposa bene con motori semplici, per cui non abbiamo bisogno di utilizzare i servizi di ASP.NET Core MVC. Resta possibile includerli con l’uso della Dependency Injection, che introdurremo successivamente nel corso del capitolo. Benché tecnicamente non sia necessario, nella maggior parte dei casi, comunque, ci troveremo di fronte a un classico controller che eredita dalla classe base `Controller`.

Proprietà e metodi di supporto della classe Controller

Come abbiamo spesso avuto modo di notare nel corso di questo libro, quando scriviamo il codice di un’applicazione web abbiamo più volte la necessità di accedere alle informazioni inerenti il contesto della richiesta. Questi dati sono contenuti all’interno dell’oggetto `HttpContext`, che è esposto attraverso la classe base `Controller`. Le principali sono riassunte nella Tabella 5.1.

Tabella 5.1 – Contesto di richiesta nella classe Controller.

Nome proprietà	Descrizione
`HttpContext`	Contiene l’istanza di HttpContext relativa alla richiesta corrente.
`ControllerContext`	Incapsula le informazioni della richiesta corrente e contiene, oltre all’HttpContext, anche informazioni relative al routing e all’istanza del controller correntemente in esecuzione.
`Request`	Incapsula il contenuto della richiesta corrente.
`Response`	Consente di accedere al flusso di risposta.
`User`	Contiene le informazioni relative all’utente corrente.

Come sappiamo, però, il compito principale di un controller è di istanziare il model e passarlo poi alla view. Quando la pagina è piuttosto semplice e non richiede la creazione di una classe ad-hoc come model, possiamo sfruttare per questo scopo le proprietà `ViewData` e `ViewBag`. Esse rappresentano un dizionario in cui possiamo memorizzare qualsiasi tipo di informazione. La differenza tra le due è costituita dal fatto che `ViewBag` è un oggetto di tipo `dynamic` e quindi ci consente di accedere alle varie chiavi memorizzate con una sintassi un po' più snella, come mostrato nell'Esempio 5.6.

Esempio 5.6

```
public IActionResult Index()
{
   // ViewData e ViewBag agiscono sul medesimo dizionario
   ViewBag.Message = "Esempio di messaggio";
   ViewData["Message"] = "Sostituisce il messaggio precedente";
   return View();
}
```

Sebbene sia comodo, questo approccio purtroppo non si rivela di certo vincente dal punto di vista della manutenibilità e della robustezza, visto che ci vincola a lavorare con degli `object` e, quindi, non c'è alcun controllo sulla type-safety in fase di compilazione. Quando, nel corso di questo capitolo, parleremo delle action, e anche nel prossimo capitolo dedicato alle view, vedremo come la soluzione di dotarsi di un model fortemente tipizzato sia sicuramente più robusta e destinata a perdurare nel tempo.

Ciclo di vita di un controller

L'aspetto in assoluto più importante in un'applicazione web è la gestione del ciclo di vita della risorsa che sarà impiegata per generare la risposta. Come abbiamo avuto modo di rimarcare più volte, il modello di ASP.NET Core MVC è molto semplice e infatti, in un controller, tipicamente possiamo inizializzare eventuali risorse nel costruttore e distruggerle tramite il metodo `Dispose`. Il tipico caso di utilizzo è quando vogliamo sfruttare un `DbContext` di Entity Framework, facendo in modo che resti attivo per tutta la durata della richiesta, come nell'Esempio 5.7.

Esempio 5.7

```
public class HomeController : Controller
{
   private NorthwindContext _context;
   public HomeController()
   {
      _context = new NorthwindContext();
   }
   protected override void Dispose(bool disposing)
   {
      if (_context != null)
         _context.Dispose();
      base.Dispose(disposing);
   }
}
```

> *In questo esempio stiamo utilizzando una particolare funzionalità, chiamata controller activator, che è responsabile di istanziare i controller quando necessario. Questo è in grado di gestire dipendenze e, pertanto, di valorizzare eventuali parametri del costruttore. L'esempio qui mostrato è in realtà poco corretto da un punto di vista pratico e vedremo come sia possibile sostituirlo con uno che faccia uso di IoC container più tardi in questo stesso capitolo.*

Oltre alla inizializzazione e distruzione del controller, abbiamo a disposizione anche i metodi illustrati nella Tabella 5.2, secondo l'ordine cronologico in cui vengono invocati, per introdurre la nostra logica personalizzata durante le varie fasi di elaborazione della richiesta, in particolare relativamente alla gestione degli errori, dell'autorizzazione o della vera e propria esecuzione della action.

Tabella 5.2 – Metodi personalizzabili della classe Controller.

Nome metodo	Descrizione
`OnActionExecutionAsync`	Invocato subito prima dell'esecuzione del codice della action, con il supporto per async.
`OnActionExecuting`	Invocato subito prima dell'esecuzione del codice della action.
`OnActionExecuted`	Invocato appena dopo l'esecuzione della action.

Sebbene possano sussistere numerose ragioni per voler ridefinire il contenuto di questi metodi, tuttavia solo raramente ci troveremo a procedere in questo senso: per questo tipo di necessità, infatti, ASP.NET Core MVC dispone di una serie di oggetti probabilmente più idonei e maggiormente versatili, denominati filtri, di cui parleremo più avanti nel capitolo, che si aggiungono ai middleware, che restano invece più orientati ad ASP.NET Core in generale.

Uso della Dependency Injection nei controller

Inversion of Control (IoC) è un principio di design che consente, come il termine stesso suggerisce, di invertire il controllo, rispetto alla programmazione procedurale, in cui il codice richiama le librerie per eseguire i task, consentendo invece al framework di richiamare il codice.

> *"High level modules should not depend on low level modules; both should depend on abstractions."*
> *- Dependency Inversion Principle*
>
> *"Moduli di alto livello non dovrebbero dipendere da moduli di basso livello; entrambi dovrebbero dipendere da astrazioni." - Principio di Dependency Inversion*

In parole povere, l'IoC consente di creare applicazioni modulari, con un grado di estendibilità maggiore. Senza l'uso di IoC, il flusso della nostra logica è staticamente espresso attraverso oggetti che sono collegati l'uno all'altro. Con l'IoC, invece, il flusso dipende dal grafo di oggetti che è costruito in seguito all'esecuzione del programma. Questo è reso possibile grazie all'uso di astrazioni (come, per esempio, le interfacce), in luogo delle implementazioni concrete. Il binding tra interfacce e oggetti è stabilito a runtime attraverso un meccanismo di dependency injection (DI) (o un service locator). Con la dependency injection non è necessario stabilire questo vincolo staticamente, a runtime, pertanto diventa possibile cambiare strategia (e quindi, logica di implementazione) semplicemente dichiarando una corrispondenza differente tra l'astrazione che utilizziamo e la sua corrispondenza.

Questo concetto è abbastanza noto a qualsiasi sviluppatore che abbia mai utilizzato un'interfaccia: queste ultime, infatti, non sono un legame forte ma un vincolo debole, cioè un contratto, che garantisce che tutte le classi che implementano un'interfaccia abbiano almeno i metodi che il relativo contratto richiede. Per il polimorfismo, diventa possibile trattare una qualsiasi classe che implementa un'interfaccia come l'interfaccia stessa, rendendo possibile uno scambio a livello di reale implementazione utilizzata. Grazie all'uso di queste tecniche, si può produrre un'applicazione le cui componenti siano scarsamente accoppiate (loose coupling).

Per capire meglio questo concetto, proviamo a implementare un'interfaccia e una classe come nell'Esempio 5.8.

Esempio 5.8

```
public interface IMyService
{
   IEnumerable<Person> GetPeople();
}

public class MyFakeService : IMyService
{
   public IEnumerable<Person> GetPeople()
   {
      for (int i = 0; i < 10; i++)
      {
         yield return new Person
         {
            FirstName = "Daniele",
            LastName = "Bochicchio " + i,
         };
      }
   }
}
```

In questo esempio abbiamo creato un semplice metodo che restituisce, con un ciclo, dieci elementi all'interno di una collection. Questo metodo può essere facilmente utilizzato all'interno di una action per farci dare l'elenco delle persone da mostrare a video, o esportare in JSON.

Tradizionalmente, se non conoscessimo i concetti di IoC e dependency injection, finiremmo per fare un'istanza della classe `MyFakeService` e legheremmo per sempre questa implementazione al nostro scenario. Dovessimo cambiarla, per agganciare il tutto, per esempio, al database o al servizio remoto da cui provengono i dati, dovremmo andare a riscrivere buona parte del codice.

ASP.NET Core include già un motore di DI che, come abbiamo accennato nel Capitolo 3, è pervasivo in tutto il runtime ed è già utilizzato in `Startup` per registrare e invocare i servizi, tra cui i middleware, come ASP.NET Core MVC stesso. Basta guardare nel metodo

ConfigureServices per iniziare a farci un'idea di come funzionerà. In particolare, è lì che andremo a registrare la dipendenza tra la nostra interfaccia e il nostro servizio.

Esempio 5.9

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
   // i nostri servizi applicativi dopo MVC
   services.AddScoped<IMyService, MyFakeService>(); // interfaccia -> classe
   services.AddScoped<PeopleContext>();//solo lifetime
}
```

Uno dei compiti di un motore di DI è quello di gestire anche il ciclo di vita (lifetime) degli oggetti che vengono iniettati. Nell'esempio, infatti, oltre a specificare la corrispondenza tra l'interfaccia e la sua implementazione, abbiamo anche aggiunto un esempio in cui per la classe PeopleContext (il contesto di Entity Framewok Core, che approfondiremo in seguito) viene gestito semplicemente il lifetime, senza passare per un'interfaccia, perché non necessaria.

Il motore di injection di ASP.NET Core MVC supporta queste modalità:

- AddTransient: i servizi con lifetime transient sono creati ogni volta che vengono richiesti: questa modalità si sposa bene con servizi stateless.
- AddScoped: i servizi con lifetime scoped vengono creati e condivisi all'interno di ogni richiesta.
- AddSingleton: i servizi con lifetime singleton vengono creati alla prima richiesta e tutte le successive continueranno a utilizzarne l'istanza, utilizzando il pattern Singleton.

Scegliere il tipo di lifetime è un'operazione che è influenzata dal tipo di oggetto che vogliamo iniettare. Alcuni servizi, come il contesto di Entity Framework o classi che implementano il Repository Pattern, funzionano con un lifetime di tipo scoped, perché il loro ciclo di vita prevede che vengano istanziati e resi disponibili per tutta la durata della richiesta. Ma come facciamo a utilizzare il servizio che abbiamo creato in precedenza? ASP.NET Core supporta diversi tipi di iniezione della dipendenza, che possiamo apprezzare nell'esempio che segue.

Esempio 5.10

```
public class HomeController : Controller
{
   private readonly IMyService myService;
   public HomeController(IMyService myService)
   {
    this.myService = myService;
    }
    public IActionResult Index()
    {
       var model = myService.GetPeople();
       return View(model);
    }
    public IActionResult Search([FromServices]IMyService svc)
    {
       var model = svc.GetPeople();
       return View(model);
    }
}
```

Nel primo caso abbiamo specificato la dipendenza nel costruttore: è il runtime stesso a capire che abbiamo bisogno che venga iniettato il servizio e, all'interno di myService, troveremo l'istanza di MyFakeService correttamente istanziato e pronto all'uso. Un'alternativa, qualora non volessimo rendere disponibile l'istanza a tutto il controller, è quello mostrato nella action Search: in questo caso, grazie all'uso dell'attributo FromServices, otterremo lo stesso effetto, ma lo scope del servizio sarà visibile solo all'interno della action, e non su tutte le action del controller, come nel caso precedente. Se avessimo più dipendenze da dover risolvere, ci basterà specificarle tutte all'interno del costruttore.

Un'altra modalità per accedere ai servizi registrati, qualora non si possa optare per una delle due, è quella di accedere alla collection RequestServices, esposta dalla classe HttpContext.

Il vantaggio dell'uso dell'interfaccia, nel nostro caso, è che per cambiare strategia (per esempio per richiedere l'elenco delle persone a un servizio remoto) non dovremo cambiare la classe MyFakeService ma crearne un'altra, che implementi sempre l'interfaccia

IMyService, e poi variare la registrazione nel metodo ConfigureServices della classe Startup, perché ovunque facciamo riferimento all'interfaccia, nel codice venga utilizzata la nuova implementazione. Ecco spiegato, con un esempio molto semplice, quanto sia comodo e proficuo adottare un motore di DI.

Come abbiamo accennato nel capitolo precedente e come vedremo nei prossimi capitoli, tutti i servizi di ASP.NET Core sono registrati in questo modo: è così, per esempio, che possiamo accedere alla configurazione, al logging e alle informazioni sull'environment corrente.

Utilizzare un motore di IoC container differente

Benché il motore di DI ASP.NET Core sia utile per la maggior parte dei casi, tradizionalmente nel modo .NET ci sono diversi IoC container che, negli anni, hanno riscosso successo, come Autofac o Ninject. Il motore di ASP.NET Core è estendibile e quindi possiamo decidere di sostituire il motore predefinito con quello di nostra preferenza. Per Autofac, dopo aver installato i package Autofac e Autofac.Extensions.DependencyInjection, dovremo cambiare il metodo ConfigureServices all'interno di Startup, così che, con una firma differente, restituisca l'istanza del container di AutoFac appena creata.

Esempio 5.11

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
   // autofac
   var containerBuilder = new ContainerBuilder();
   containerBuilder.RegisterModule<DefaultModule>();
   containerBuilder.Populate(services);
   var container = containerBuilder.Build();
   return new AutofacServiceProvider(container);
}
```

All'interno di DefaultModule, come prevede AutoFac, andremo a registrare le nostre dipendenze, creando una classe che derivi da Module e, sulla falsariga di quanto fatto in precedenza, imposti il lifetime dei vari oggetti che andremo a utilizzare. Tutto il resto che abbiamo visto in precedenza non cambia, l'unica cosa che avremo fatto è quella di bypassare completamente il motore di IoC presente in ASP.NET Core, per utilizzare AutoFac (o uno qualsiasi degli altri supportati).

Finora abbiamo visto alcune peculiarità specifiche della classe Controller ma non abbiamo ancora affrontato il principale concetto che questo tipo introduce: stiamo parlando delle action, che saranno l'argomento della prossima sezione.

La action come gestore della richiesta

Se fossimo costretti a realizzare controller implementando di volta in volta l'interfaccia IController, saremmo obbligati a gestire manualmente la request e a interpretare quindi dall'URL quale è l'effettiva informazione richiesta dall'utente; a questo punto potremmo finalmente generare – sempre e rigorosamente a mano – la risposta da restituire.

Indubbiamente si tratta di un approccio poco pratico e probabilmente a queste condizioni ASP.NET Core MVC non avrebbe ottenuto il successo che invece oggi ha.

Il vantaggio principale di ereditare i nostri controller dalla classe Controller è che possiamo lavorare a un livello più alto di astrazione, grazie al concetto di action. Abbiamo visto che esiste già, a livello di routing, un parametro con questo nome; il compito della classe base è quello di eseguire, in base al suo valore, un metodo omonimo all'interno del nostro controller. Il risultato è che per gestire una richiesta a http://localhost/Home/Index possiamo limitarci a scrivere il codice dell'Esempio 5.12.

Esempio 5.12

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      return View();
   }
   public IActionResult ActionWithParams(int id, string search,
      int value = 0)
   {
      // con il routing standard:
      // {controller}/{action}/{id}
      // id -> route data
      // search e value -> query string o form
      return View();
```

```
    }
}
```

Come ogni metodo, anche una action può accettare dei parametri, come possiamo vedere nell'esempio precedente, nel metodo `ActionWithParams`. Essi verranno popolati automaticamente dal runtime, in base al contenuto della richiesta. Per esempio, immaginiamo di invocare la action dell'esempio precedente tramite l'URL http://localhost/ActionWithParams/5?search=test. Con le impostazioni di default per il routing, `id` verrà recuperato dal corrispondente parametro sulla route e quindi varrà `5`, mentre `search` proverrà dalla query string e sarà valorizzato a `test`. Quando abbiamo parametri facoltativi, dobbiamo poi prestare particolare attenzione, soprattutto quando, come nel caso di `value`, sono di un tipo che non ammette valori `null` (per esempio `int`): la soluzione è fornire un valore di default, come abbiamo fatto nell'esempio precedente, o utilizzare un nullable value type come `int?`.

Il risultato dell'invocazione di una action è sempre un oggetto di tipo `IActionResult`. Si tratta di una interfaccia, implementata da diverse classi di ASP.NET Core MVC che modellano le diverse tipologie di risposta restituita. Nelle prossime pagine le analizzeremo nel dettaglio.

L'interfaccia IActionResult e i diversi tipi di risposta

Negli esempi che abbiamo condotto fino a questo momento, abbiamo sempre sfruttato una action per restituire un oggetto di tipo view. In termini generali, però, non esiste alcun vincolo da parte di ASP. NET Core MVC sul fatto che debba essere sempre questo il risultato di una request, ed è per questa ragione che, nella sua forma standard, si preferisce indicare come tipo restituito un generico tipo che implementi l'interfaccia `IActionResult`.

`IActionResult` è una interfaccia che definisce il solo metodo `ExecuteResultAsync` e che si presta a rappresentare tutte le tipologie di risposta che una action può fornire, siano esse view, codice JavaScript, status code HTTP oppure oggetti serializzati in JSON. Per ognuno di questi, infatti, esiste uno specifico oggetto, che implementa questo metodo in maniera differente. Per semplificare la definizione di queste funzionalità, esiste una classe astratta, denominata `ActionResult`, che implementa questa interfaccia e definisce due entry point, aggiungendo anche un `ExecuteResult` che consente di definire l'esecuzione non async del codice, in quegli scenari se questo dovesse avere senso.

La classe `Controller`, dal suo canto, espone anche una serie di metodi helper per istanziare i vari tipi di risultato: uno di questi, per esempio, è il metodo `View`, che finora abbiamo utilizzato per ritornare delle istanze di `ViewResult`.

> *In generale, è possibile costruire una action che restituisca anche un semplice oggetto, come una stringa o un numero intero. In questo caso sarà il framework stesso a creare un risultato wrapper, di tipo ContentResult o JsonResult, che conterrà come risposta la rappresentazione tramite il metodo ToString dell'oggetto stesso (se semplice) o la sua serializzazione in formato JSON. Questo è il meccanismo utilizzato anche dai controller POCO per generare l'output, ma consente di avere nello stesso controller anche action che restituiscano il risultato in formato JSON, così da essere più facilmente utilizzabili come endpoint in applicazioni SPA/AJAX.*

Uno dei principali vantaggi di questo approccio è costituito anche dalla sua espandibilità: nel caso sia necessario, infatti, possiamo creare il nostro tipo di risultato personalizzato, semplicemente ereditando da questa classe base. Per il momento, però, concentriamoci su quanto di standard viene fornito dal framework, iniziando in particolare dal tipo più utilizzato, ossia `ViewResult`.

I tipi ViewResult e PartialViewResult

Quando una action ritorna un'istanza di `ViewResult`, la risposta inoltrata al chiamante è nient'altro che una pagina HTML. Il modo più semplice per costruirla è sfruttare il metodo `View` di un controller, come abbiamo visto nel precedente Esempio 5.12, restituendo così la view predefinita della action, ossia quella indicata dalla convenzione, che deve avere lo stesso nome del file e che si trovi nella directory `Views\[NomeController]` o all'interno di `Views\Shared`. Nell'esempio che abbiamo citato, quindi, la view corrisponderà al file `Views\Home\Index.cshtml`.

> *Anche questo è un comportamento del tutto personalizzabile e dettato dall'implementazione standard del view engine di ASP.NET Core MVC.*

In linea generale, però, una action può restituire, a seconda dei casi, view differenti: per esempio, potremmo visualizzare una pagina di errore quando il salvataggio di un dato non è andato a buon fine, oppure mostrare il nuovo record inserito. Per queste evenienze, esiste un overload del metodo `View` che ci permette di specificarne il nome, come nell'Esempio 5.13.

Esempio 5.13

```
public IActionResult About()
{
    if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
```

```
      return View("SundayAbout");
   return View();
}
```

Il codice precedente, anche se molto banale, restituisce una view differente nel caso in cui venga invocato di domenica.

In tutti gli esempi visti fino a questo momento, non ci siamo mai preoccupati della modalità con cui possiamo passare delle informazioni alle varie view: sappiamo che il pattern MVC prevede la figura del model come contenitore di questo tipo di dati ma noi, per il momento, ci siamo sempre limitati a sfruttare `ViewBag` e `ViewData` per questo scopo. In un'applicazione reale, però, non è pensabile di procedere in questi termini ma si preferisce sfruttare come model degli oggetti appositi, come nel caso illustrato nell'Esempio 5.14, in cui abbiamo passato alla view un oggetto `Person` come model.

Esempio 5.14

```
public IActionResult Search(int id)
{
   var model = new Person()
   {
      FirstName = "Daniele",
      LastName = "Bochicchio"
   };
   return View(model);
}
```

Il vantaggio di questo approccio risiede fondamentalmente nel fatto che ci aiuta a non sbagliare: come vedremo nel prossimo capitolo, dedicato alla view, infatti, Visual Studio è in grado di supportarci nella scrittura del codice grazie all'intellisense, e di segnalarci eventuali errori (come l'accesso a un membro non valido) direttamente nella finestra dell'editor. Inoltre, abilitando la compilazione delle view, diventa possibile ricevere una notifica in tal senso anche a compile time, anziché a runtime, evitando spiacevoli errori che si manifestano solo navigando verso un certo URL.

Simili al tipo `ViewResult` e all'helper `View`, esistono anche il tipo `PartialViewResult` e il corrispondente helper `PartialView` che, a differenza di quanto abbiamo visto finora, restituiscono una view priva della corrispondente layout page, di cui parleremo in maggior dettaglio nel prossimo capitolo. Per il momento, ci basti pensare che una layout page è semplicemente una pagina base che le view possono utilizzare per mantenere un look omogeneo all'interno di un sito.

Esempio 5.15

```
public IActionResult Contact(bool embed = false)
{
   if (embed)
      return PartialView();
   else
      return View();
}
```

Questo approccio è comodo quando vogliamo farci dare l'HTML di una risposta senza tutto il contorno che ci sarebbe. Nell'esempio precedente sarà sufficiente utilizzare il parametro per farci restituire solo l'HTML prodotto dalla view, bypassando la layout page.

I tipi RedirectResult, RedirectToRouteResult e HttpStatusCodeResult

A volte la risposta di una action non è un contenuto di qualche tipo ma un semplice redirect verso un altro indirizzo. Il tipo `RedirectResult` ci consente di reindirizzare il browser verso un indirizzo arbitrario, come viene mostrato nell'Esempio 5.16.

Esempio 5.16

```
public IActionResult GoToGoogle()
{
   return Redirect("http://www.aspitalia.com");
}
```

Nel codice precedente, abbiamo usato il metodo `Redirect` per reindirizzare l'utente sul sito di ASPItalia.com, restituendo uno status code 302 (temporary redirect). Se necessario, è disponibile anche il metodo `RedirectPermanent`, che restituisce uno status code 301 (permanent redirect).

Quando dobbiamo rimandare a un'altra action del nostro sito, però, è molto più comodo utilizzare il metodo `RedirectToAction`, che ci permette di generare un URL in base ai parametri di routing.

Esempio 5.17

```
public IActionResult BackToIndex()
{
```

```
   return RedirectToAction(nameof(Index));
}

public IActionResult ComplexRedirect()
{

   return RedirectToAction(
      nameof (Details),
      nameof(CustomersController).Replace("Controller", ""),
      new { id = 5 });
}
```

L'Esempio 5.17 mostra un paio di tipologie di utilizzo possibili per questo metodo, che restituisce un oggetto di tipo `RedirectToRouteResult`. Il primo caso rimanda alla action `Index` del medesimo controller, mentre la seconda chiamata è più complessa e genera un URL del tipo `http://localhost/Customers/Details/5`. Anche per quanto riguarda questo metodo, è disponibile un redirect permanente tramite il metodo `RedirectToActionPermanent`.

Questi metodi accettano delle stringhe come parametri e, grazie all'uso di `nameof`, introdotto con C# 6, diventa possibile referenziare direttamente il nome di metodi e proprietà, invece di dichiarare direttamente una stringa. Questo rende molto più semplice rinominare un membro (o un controller) senza lasciare stringhe non più valide all'interno.

> *L'uso di nameof con i controller necessita che la stringa venga ripulita del suffisso controller, presente per convenzione. In un'applicazione reale questo replace in linea può essere sostituito dall'uso di un extension method. Un altro approccio è quello di non specificare il suffisso Controller e dare un nome più semplice ai controller, aggiungendo l'attributo [Controller] alla classe.*

In altre occasioni, invece, può essere necessario impostare un particolare status code per la risposta. Per esempio, se stiamo cercando un `Person` il cui id è inesistente, è corretto restituire uno status code 404 (not found). Per questo tipo di necessità, possiamo sfruttare l'helper dell'Esempio 5.18.

Esempio 5.18

```
public IActionResult Find(int id)
{
   Person person = null; // aggiungere la ricerca su database...
   if (person == null)
      return HttpNotFound();
   return View(person);
}
```

La action precedente restituisce un `HttpStatusCodeResult` che abbiamo creato tramite il metodo `HttpNotFound`. Ovviamente, possiamo anche costruire manualmente un'istanza di questo oggetto, per ritornare un qualsiasi status code, se necessario.

Restituire JSON con JsonResult

Nello sviluppo client side, spesso abbiamo la necessità di restituire un certo dato serializzato in formato JSON. Anche per questa evenienza è disponibile un oggetto apposito – nella fattispecie `JsonResult` – che può essere istanziato come nell'Esempio 5.19.

Esempio 5.19

```
public IActionResult GetPerson(string firstName, string lastName)
{
   var result = new Person
   {
      FirstName = firstName,
      LastName = lastName
   };
   return Json(result, JsonRequestBehavior.AllowGet);
}
```

Questo metodo espone diversi overload, che permettono di specificare parametri addizionali, quali il content-type da restituire e l'encoding da utilizzare. Un aspetto da segnalare è che, per default, una action di questo tipo è invocabile solo con una chiamata diversa da GET, come POST, PUT o DELETE, per esempio; anche per quanto riguarda questo aspetto, è disponibile un overload che ci permette di abilitare anche il verbo GET, utilizzando `JsonRequestBehavior.AllowGet`.

Come detto, possiamo restituire JSON facendo fare direttamente il lavoro alla action, specificando opportunamente il tipo di ritorno, come viene specificato nell'Esempio 5.20.

Esempio 5.20

```
public Person GetPerson(string firstName, string lastName)
{
   return new Person
   {
      FirstName = firstName,
      LastName = lastName
   };
}
```

Come abbiamo detto, in questo caso sarà restituita direttamente la serializzazione in format JSON dell'oggetto specificato (valido anche per collection).

Il tipo FileResult e il metodo File

Questo tipo di action result è utile tutte le volte in cui vogliamo restituire un file per il download. Può essere istanziato tramite il metodo `File` della classe `controller`, come nell'Esempio 5.21.

Esempio 5.21

```
public IActionResult Download()
{
   return File("file.xlsx", "application/excel", "export.xlsx");
}
```

Nel codice precedente abbiamo utilizzato come sorgente dati il file system, specificando il path completo del file, e indicando il mime type e il nome del file che vogliamo restituire al browser. Esistono anche ulteriori overload, che permettono di utilizzare come sorgente un array di `byte` o un qualsiasi stream, per i casi in cui il contenuto dovesse essere calcolato a runtime, piuttosto che generato a partire da un database.

Il tipo ContentResult e il metodo Content

Le diverse tipologie di `ActionResult` che abbiamo avuto modo di esplorare fino a questo momento sono tutte relative a uno specifico risultato. Alle volte, invece, abbiamo bisogno di una maggiore versatilità, vogliamo cioè controllare esattamente il contenuto della risposta; per questo scopo, abbiamo a disposizione il tipo `ContentResult`, che può essere facilmente generato a partire dal metodo `Content`, specificando anche l'encoding e il mime type desiderato, come nell'Esempio 5.22.

Esempio 5.22

```
public IActionResult GetSomeText()
{
   string result = "Lorem ipsum...";
   return Content(result, "text/plain", Encoding.UTF8);
}
```

Queste che abbiamo esaminato sono le differenti tipologie di risultato fornite, out-of-the-box, da ASP.NET Core MVC. In realtà, come abbiamo già accennato, è possibile creare un tipo personalizzato e sfruttarlo analogamente a quelli che abbiamo già visto; questo, assieme ad altri, sarà uno degli aspetti di espandibilità che tratteremo nei Capitoli 13 e 14. Per il momento, continuiamo a concentrarci sulle funzionalità standard, in particolare a proposito delle modalità per controllare l'esecuzione delle action all'interno dei nostri controller.

Controllo dell'esecuzione di una action

Finora tutti gli esempi che abbiamo citato hanno come punto in comune il fatto che le action rispondano al relativo URL a prescindere dal verb HTTP utilizzato per la richiesta; un'altra costante è rappresentata dal fatto che a ogni metodo pubblico corrisponda una action del medesimo nome. Questi aspetti rappresentano solo i comportamenti di default, che possono essere però in qualche modo controllati, come nell'Esempio 5.23.

Esempio 5.23

```
[HttpGet]
public IActionResult GetOnly()
{
   return View();
}

[HttpPost]
```

```
public IActionResult PostOnly()
{
   return View();
}

[ActionName("ActualName")]
public IActionResult ThisIsNotTheName()
{
   return View();
}

[NonAction]
public object NotAnAction()
{
   return null;
}
```

Nel codice precedente possiamo notare come, grazie all'uso degli attributi `HttpGetAttribute` e `HttpPostAttribute`, siamo in grado di circoscrivere l'esecuzione di una action, rispettivamente, alle sole richieste in GET o POST; esistono attributi simili che corrispondono a tutti i verb HTTP.

Il terzo metodo che abbiamo creato, che abbiamo chiamato `ThisIsNotTheName`, corrisponde invece a un caso in cui il nome del metodo differisce da quello della action corrispondente, grazie all'attributo `ActionNameAttribute`. Infine, se utilizziamo `NonActionAttribute`, possiamo far sì che un metodo pubblico non sia esposto come action, e quindi non possa essere invocato tramite il relativo URL. Questo esempio, in particolare, ritorna molto comodo con i controller POCO.

Esecuzione asincrona di una action

L'esecuzione di codice asincrono può apportare benefici alla scalabilità e alle performance del sistema: in realtà, tutto il codice sincrono è, di fatto, asincrono, perché tale è il comportamento dei sistemi operativi e del network su cui si basano. Di fatto, scrivere codice sincrono era molto più semplice, prima dell'avvento di `async/await` in C# 5, poiché l'alternativa era fare uso degli eventi, che frammentavano la logica del codice.

In questo caso, infatti, quando una action ha la necessità di effettuare un'invocazione remota, sia essa a un web service, a un device, al file system o a un database, se il codice viene eseguito in maniera sincrona il risultato è che il thread di esecuzione resta bloccato, in attesa del completamento dell'operazione. Al contrario, sfruttando il codice asincrono, il vantaggio è che tale thread viene restituito al pool e quindi è potenzialmente utilizzabile per servire un'altra richiesta, portando benefici al sistema nel suo complesso. Al termine dell'operazione asincrona, il runtime si occuperà automaticamente di recuperare un altro thread tra quelli disponibili, di effettuare il marshalling del `HttpContext` e, finalmente, di proseguire con l'esecuzione del codice della action.

Realizzare codice asincrono in ASP.NET Core MVC è assolutamente analogo a realizzarlo in ASP.NET MVC, poiché entrambi si basano sul pattern async/await di C#. Riprendendo un classico esempio che scarica un contenuto da remoto, il primo passo è di aggiungere la parola chiave `async` alla action, come nell'Esempio 5.24.

Esempio 5.24

```
public async Task<IActionResult> AsyncAction()
{
   WebClient client = new WebClient();
   string s = await
      client.DownloadStringTaskAsync("http://www.servizio.remoto");
   return Content(s);
}
```

Come ogni metodo asincrono, l'unico vincolo che abbiamo è quello di utilizzare un `Task` e, nello specifico, un `Task<IActionResult>` come tipo di ritorno. A questo punto, non ci resta che utilizzare la parola chiave `await` per invocare il metodo asincrono desiderato, per esempio `DownloadStringTaskAsync` nel codice precedente, per far sì che possiamo sfruttare questo paradigma nella nostra applicazione web.

Quando un metodo restituisce il tipo `Task` o `Task<T>`, l'uso di una action asincrona porta benefici in termini di scalabilità e andrebbe sempre attuato.

> *Il pattern async/await è alla base delle ultime versioni di C#. Benché fuori dagli scopi di questo libro, consigliamo di approfondire l'argomento attraverso la lettura di questo articolo su: http://aspit.co/ai9.*

Finora abbiamo visto come, seppure molto semplice, l'architettura di controller e action sia uno strumento estremamente flessibile, utile per gestire molteplici casistiche. Abbiamo però, su questo fronte, ancora un piccolo tassello da inserire nel mosaico, che ci permetterà di iniettare della logica durante le varie fasi del ciclo di elaborazione: i filtri.

L'infrastruttura dei filtri

In ASP.NET esiste da sempre un concetto che gli sviluppatori hanno apprezzato, cioè quello degli HttpModule, tramite i quali possiamo intercettare gli eventi che si susseguono durante l'elaborazione della richiesta. Abbiamo visto, nel Capitolo 3, come ASP.NET Core cambi questo approccio, attraverso l'introduzione dei middleware, che sono più flessibili e più potenti, grazie alla loro architettura.

Gli HttpModule, ovviamente, non sono più disponibili anche nell'ambito di un progetto ASP.NET Core MVC, ma abbiamo ancora una particolare tipologia di oggetti, per certi versi simili ai moduli, ma che hanno il pregio di essere maggiormente calati nell'architettura di ASP.NET Core MVC: stiamo parlando dei filtri.

Dal punto di vista strettamente pratico, si tratta di classi che implementano una serie d'interfacce, ognuna specifica per una particolare funzionalità. Distinguiamo quattro tipologie di filtri:

- Action filter: implementano l'interfaccia `IActionFilter` e ci consentono di gestire le fasi immediatamente precedenti e successive all'esecuzione della action.
- Result filter: simili ai precedenti, implementano l'interfaccia `IResponseFilter` e ci danno la possibilità di iniettare codice nelle fasi immediatamente precedenti e successive all'esecuzione del result di una action.
- Exception filter: implementano `IExceptionFilter` e contengono codice che gestisce situazioni di errore, permettendo di intercettare un'eccezione non gestita sul server.
- Authorization filter: implementano `IAuthorizationFilter` e ci permettono di gestire le policy secondo cui consentire l'esecuzione di una determinata action; parleremo in maniera piuttosto estesa di questi oggetti nel Capitolo 18, dedicato agli aspetti di security.

La classe Controller implementa solo l'interfaccia IActionFilter (oltre che IAsyncActionFilter) e, quindi, è a tutti gli effetti un filtro; questa classe ci consente di gestire le varie casistiche tramite una serie di metodi virtuali, che abbiamo visto nelle prime pagine di questo capitolo. Rispetto ad ASP.NET MVC, in ASP.NET Core MVC la classe controller non implementa le altre interfacce, perché l'infrastruttura dei middleware è abbastanza sovrapponibile ai filtri e questi servizi, più generici, non sono implementati con questo approccio.

Per capire come utilizzare questi strumenti, cerchiamo di calarli in un contesto pratico, quindi, per esempio, immaginiamo di voler forzare l'esecuzione di una determinata action solo in un contesto HTTPS. Per questo scopo, ASP.NET Core MVC espone la classe `RequireHttpsAttribute`. Come è facilmente intuibile dal nome, si tratta di un attributo, che può essere utilizzato come nell'Esempio 5.25.

Esempio 5.25

```
public class HomeController : Controller
{
   [RequireHttps]
   public IActionResult SecureAction()
   {
      return Content("Secure content");
   }
}
[RequireHttps]
public class SecureController : Controller
{
   public IActionResult Index()
   {
      return Content("Secure content");
   }
}
```

L'effetto di questo filtro è di compiere in automatico un redirect verso lo stesso URL, ma in HTTPS. Dal codice precedente, possiamo notare la grande versatilità che gli attributi possono offrirci: possiamo infatti decorare sia una singola action sia un intero controller, proteggendone quindi tutte le action.

Il concetto di filtri può sembrare un po' limitato per quanto abbiamo visto finora, ma ciò dipende dal fatto che ci siamo limitati a considerare i pochi filtri standard inclusi in ASP.NET Core MVC. Il grande vantaggio di questo strumento, invece, è la possibilità di crearne di personalizzati. Ad ogni modo, i filtri sono abbastanza sovrapponibili ai middleware in termini di funzionalità, per cui cosa scegliere? La risposta è semplice: i filtri sono un'estensione solo di ASP.NET Core MVC, mentre i middleware si applicano all'intera pipeline, quindi a qualsiasi altro tipo di richiesta/risposta. Generalmente si scelgono i filtri quando la logica da applicare è valida solo per i controller e non è

necessario costruire un middleware, che è anche più complesso da realizzare, oltre che maggiormente invasivo, perché applicato a tutte le richieste (fatto salvo il caso in cui si registri un middleware come filtro, come vedremo nel Capitolo 14).

Conclusioni

Dopo aver dato una overview del modello di ASP.NET Core MVC, nel capitolo precedente, in questo abbiamo iniziato a esplorare a fondo le funzionalità di questo framework, occupandoci in particolare dei controller.

Innanzitutto, abbiamo visto come il flusso della risposta tragga origine dal routing, tramite il quale vengono individuati il controller e la action che dovranno essere eseguiti. Successivamente abbiamo dato un'occhiata alla classe `Controller`, dalla quale tutti i nostri controller tipicamente ereditano, e alle funzionalità basilari che essa espone. Tra queste, il principale concetto introdotto da questa classe è quello di action, ossia del metodo in grado di processare la richiesta e di restituire il relativo `IActionResult`.

Da questo tipo ereditano i vari tipi di risposta che una action può fornire: abbiamo mostrato i vari oggetti già forniti out-of-the-box da ASP.NET Core MVC, che modellano diversi tipi di risultato, dalle view, al JSON, fino ad arrivare a file o status code HTTP. Infine, come ultimo argomento, abbiamo introdotto il concetto dei filtri, tramite cui possiamo personalizzare la logica del flusso di risposta, iniettando del codice in alcuni punti chiave.

Nel prossimo capitolo ci occuperemo dell'ultimo tassello del pattern MVC, chiamato in causa per generare l'HTML che sarà poi inviato all'utente: inizieremo il nostro viaggio alla scoperta delle view.

6
Le view e Razor

Nello scorso capitolo abbiamo approfondito il ruolo che i controller e le action hanno nell'ambito di un'applicazione ASP.NET Core MVC nell'interpretare le varie richieste che arrivano al server e generare il tipo di risposta più opportuno. Al di là delle numerose possibilità che abbiamo, il risultato più comune per una action dovrà essere markup HTML e il componente incaricato di generare markup in ASP.NET Core MVC è una view.

In questo capitolo introdurremo innanzi tutto Razor, ossia l'engine tramite il quale potremo scrivere il codice delle view, e vedremo come la sua sintassi ci permetta di realizzare dei template in maniera parecchio versatile e leggibile. Successivamente ci occuperemo di tutti quegli strumenti che ci vengono forniti dal framework per rendere il più agevole possibile il nostro lavoro: in particolare parleremo delle layout view, tramite i quali possiamo dare un look uniforme alle pagine del nostro sito e supportare diverse tipologie di device, realizzando interfacce ad-hoc per ognuno di essi.

Poi introdurremo il concetto di HTML e tag helper, lo strumento proposto da ASP.NET Core MVC che ci consente di scrivere il markup delle pagine lavorando a un livello più alto di astrazione: in particolare, introdurremo il funzionamento di alcuni di essi (molti helper riguardano in maniera più specifica la realizzazione di form di input e, pertanto, li ritroveremo nel capitolo successivo), con un particolare occhio anche a partial view e child action, grazie alle quali possiamo creare componenti dell'interfaccia riutilizzabili in molte pagine.

Creare view in ASP.NET Core MVC grazie a Razor

Nel capitolo precedente abbiamo accennato al fatto che una view è quel componente del pattern MVC che ha il compito, nel caso di un'applicazione web, di rappresentare il model tramite markup HTML, in modo che poi possa essere servito al browser dell'utente. Dal punto di vista strettamente pratico, però, una view è un file posizionato all'interno di una particolare strutture di directory, che abbiamo già avuto modo di vedere ma che, per completezza, riportiamo anche nella Figura 6.1.

Questi file, al loro interno, devono conciliare l'esistenza sia del markup sia del codice tramite cui elaborare il contenuto del model stesso, e pertanto hanno un'estensione particolare (`.cshtml`) e sono scritti in una sintassi specifica che prende il nome dal view engine che sarà poi in grado di processarli, ossia Razor. Inizialmente introdotto con ASP.NET MVC, è disponibile in una versione aggiornata (e potenziata, come vedremo) per ASP.NET Core MVC.

Si tratta a tutti gli effetti di un nuovo linguaggio, per cui cercheremo di partire dalla sintassi di base per poi arrivare a realizzare pagine via via più complesse.

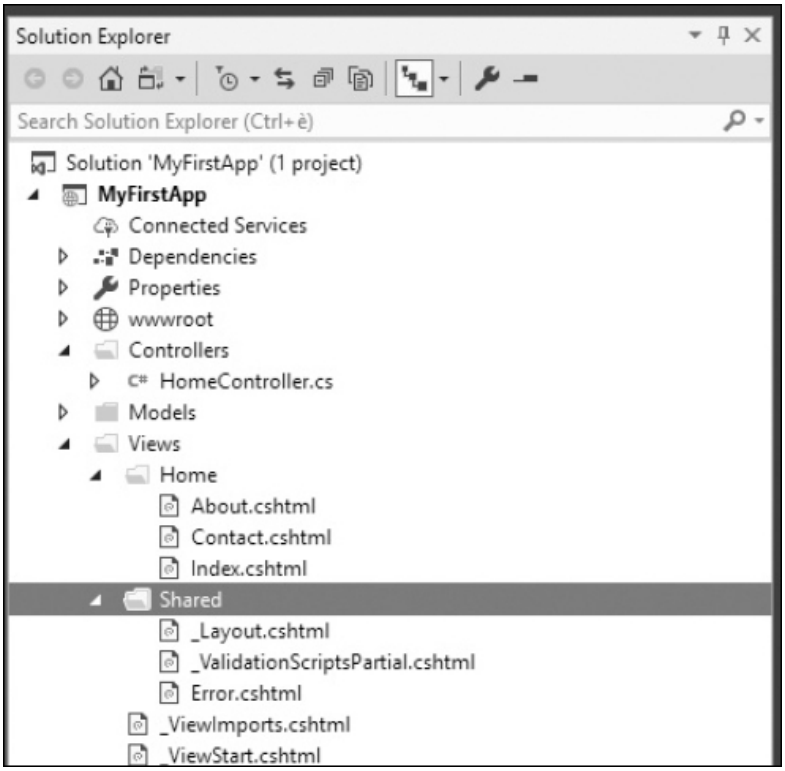


Figura 6.1 – Struttura delle directory per le view di un progetto ASP.NET Core MVC.

La sintassi di base

Visto che all'interno di una view devono coesistere sia codice sia markup, il compito di un view engine come Razor è innanzitutto quello di poter contraddistinguere questi due contesti, così che possiamo descrivere la logica secondo cui la nostra pagina dovrà essere generata. Il passaggio dal contesto di markup al contesto del codice (o viceversa) prende il nome di context switching e, nel caso di Razor, avviene grazie al carattere @ per passare da markup a codice e tag (o `@:`) per passare da codice a markup.

In particolare, in una view, possiamo definire dei blocchi di codice tramite le parole chiave `@{ ... }` in C#, come viene mostrato nell'Esempio 6.1.

Esempio 6.1

```
@{
   ViewBag.Title = "Index";
   var myString = "Una stringa di esempio";
   int someValue = 24;
}

<div>
   <h1>Titolo pagina</h1>
   <p>Lorem ipsum dolor sit amet...</p>
</div>
@ {
   //qui C#
   <div>Qui markup</div>
   @: questo è ancora markup
   //qui di nuovo C#
}
```

Questi blocchi sono utili perché ci permettono di dichiarare delle variabili che poi saranno visibili all'interno del codice dell'intera view.

Di fianco al codice, ovviamente, possiamo includere anche del markup HTML, come possiamo notare nell'esempio in alto, senza dover prendere alcun accorgimento particolare. All'interno del markup, poi, possiamo effettuare all'occorrenza un nuovo context switching, ancora una volta utilizzando il carattere @; fintanto che restiamo all'interno di una singola riga, non è necessario che creiamo un nuovo blocco e possiamo sfruttare la sintassi inline dell'Esempio 6.2.

Esempio 6.2

```
<p>Oggi è @DateTime.Today</p>

<p>Indirizzo email: daniele@aspitalia.com</p>

<p>Seguimi su Twitter: @@dbochicchio</p>

<p style="font-size:@(someValue)px">Testo grande</p>

@*Questo è un commento*@
```

L'engine è abbastanza scaltro da distinguere i casi in cui il carattere @ è inserito per altre finalità, come nel caso di un indirizzo email. Ovviamente non è detto che il context switching avvenga necessariamente nel contenuto di un tag. La terza riga del codice precedente mostra come possiamo sfruttare la variabile che abbiamo definito nell'Esempio 6.1 come componente dell'attributo style. Se necessario, possiamo isolare esplicitamente il blocco di codice che vogliamo passare a Razor, sfruttando le parentesi tonde, come abbiamo fatto nel codice precedente per separare il testo *"px"* dal nome della variabile `someValue`. L'uso invece di `@* ... *@` ci permette di inserire commenti nel codice C#, che non appariranno nel markup, dato che sono commenti server-side.

Branch e cicli

Tipicamente, quando realizziamo una view, abbiamo anche la necessità di visualizzare delle porzioni di pagina al verificarsi di una determinata condizione, o di ripetere lo stesso markup più volte. Per queste necessità possiamo sfruttare i blocchi `if`, `for` e `foreach`, o qualsiasi altro statement, come nell'Esempio 6.3.

Esempio 6.3

```
@if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
{
   <div>Buona domenica</div>
}
else
{
   <p>Oggi è @DateTime.Today</p>
}
```

```
<ul>
   @for (int index = 0; index < 7; index++)
   {
      <li>@((DayOfWeek)index)</li>
   }
</ul>
```

La sintassi da utilizzare è piuttosto intuitiva, e sfrutta le caratteristiche di C#, con Razor che è in grado di individuare autonomamente i vari tag e considerarli pertanto come tali. Quando non abbiamo tag da inserire, possiamo sfruttare il tag speciale `<text>` dell'Esempio 6.4.

Esempio 6.4

```
@if (DateTime.Today.DayOfWeek == DayOfWeek.Sunday)
{
   <text>Buona domenica</text>
}
```

Questo tag viene ignorato in fase di rendering, ma serve a segnalare un contesto di markup al parser di Razor. Differentemente, il codice verrebbe considerato un'istruzione lato server (dopo la prima parantesi graffa) e pertanto avremmo un errore a runtime, poiché la stringa che segue non è compilabile in C#.

Definire funzioni in una view

Abbiamo già visto che, tramite il blocco `@{ ... }` possiamo dichiarare variabili che saranno visibili all'intera view. Alle volte può essere utile anche definire delle funzioni, e per questa necessità si usa il blocco `@functions` dell'Esempio 6.5.

Esempio 6.5

```
@functions {
      private string FormatDate(DateTime source)
      {
         return source.ToShortDateString();
      }
}

<p>Oggi è @FormatDate(DateTime.Today)</p>
```

Analogamente al caso delle variabili, anche queste funzioni saranno visibili all'interno della view e quindi possono essere utili per centralizzare parte della logica e riutilizzarla in più punti.

Le view e il model: tipizzazione debole e forte

In tutti gli esempi che abbiamo realizzato fino a questo punto del libro, il model ha avuto un'importanza davvero marginale: tutte le volte che abbiamo avuto la necessità di passare informazioni dal controller alla view, abbiamo sfruttato l'oggetto `ViewBag` (o `ViewData`), ossia un `Dictionary` condiviso tra questi due componenti del pattern MVC.

In realtà, questo modo di procedere è davvero poco pratico nel momento in cui ci troviamo a realizzare applicazioni reali: usando questi strumenti, infatti, non abbiamo alcun ausilio dal tool di sviluppo, né dal punto di vista della correttezza delle chiavi che stiamo utilizzando, né dal punto di vista della tipizzazione. Consideriamo l'Esempio 6.6.

Esempio 6.6

```
public IActionResult Wrong()
{
   ViewBag.Today = "non è una data";
   return View();
}
```

Esempio 6.6 – View

```
<h2>@ViewBag.Today.ToShortDateString()</h2>
```

Sebbene il codice precedente sia palesemente errato, Visual Studio non è in grado di darci alcun tipo di avviso durante la scrittura, proprio in virtù del fatto che stiamo sfruttando il late binding, ossia stiamo rimandando al runtime il controllo della congruità dei tipi.

Il risultato è che, se proviamo a visualizzare la pagina, otteniamo l'errore di runtime mostrato nella Figura 6.2.

An unhandled exception occurred while processing the request.

RuntimeBinderException: 'string' does not contain a definition for 'ToShortDateString'

CallSite.Target(Closure , CallSite , object)

Stack Query Cookies Headers

RuntimeBinderException: 'string' does not contain a definition for 'ToShortDateString'

```
CallSite.Target(Closure , CallSite , object )
System.Dynamic.UpdateDelegates.UpdateAndExecute1<T0, TRet>(CallSite site, T0 arg0)
AspNetCore._Views_Home_Wrong_cshtml+<ExecuteAsync>d__0.MoveNext() in Wrong.cshtml
+       6. <h2>@ViewBag.Today.ToShortDateString()</h2>
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderPageCoreAsync>d__16.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderPageAsync>d__15.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
System.Runtime.CompilerServices.TaskAwaiter.GetResult()
Microsoft.AspNetCore.Mvc.Razor.RazorView+<RenderAsync>d__14.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewExecutor+<ExecuteAsync>d__22.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewFeatures.ViewExecutor+<ExecuteAsync>d__21.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
Microsoft.AspNetCore.Mvc.ViewResult+<ExecuteResultAsync>d__26.MoveNext()
System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
```

Figura 6.2 – Errore a runtime dovuto alla mancata tipizzazione.

La soluzione sicuramente più manutenibile e sicura è quella di realizzare un model, che mantenga al suo interno tutte le informazioni da visualizzare, in maniera tipizzata. ASP.NET Core MVC non pone particolari vincoli alle loro realizzazione, anche se per convenzione è preferibile inserirli nella directory `Models` e denominarli con il suffisso *-ViewModel*. Il model per la home page è quello dell'Esempio 6.7.

Esempio 6.7

```
public class HomeViewModel
{
   public string Title { get; set; }
   public DateTime TheDate { get; set; }
}
```

A questo punto non dobbiamo far altro che creare un'istanza di questa classe all'interno della action e passarla come parametro alla view, come nell'Esempio 6.8.

Esempio 6.8

```
public IActionResult TypedAction()
{
   var model = new HomeViewModel()
   {
      Title = "Action tipizzata",
      TheDate = DateTime.Today
   };
   return View(model);
}
```

Affinché possiamo sfruttare `HomeViewModel` anche nel codice della view, dobbiamo creare quella che, in ASP.NET Core MVC, prende il nome di view tipizzata, grazie all'apposito checkbox della finestra di dialogo della Figura 6.3. In particolare, Visual Studio ci permette anche di selezionare il tipo che vogliamo utilizzare tramite una combobox.

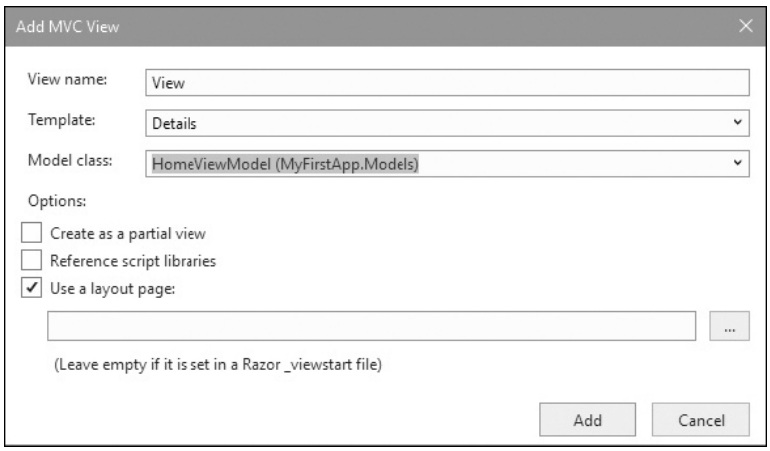


Figura 6.3 – Finestra di dialogo per creare una view tipizzata.

Il risultato di questa operazione è una nuova view che, a differenza delle precedenti che abbiamo creato finora, presenta la direttiva `@model` che indica il tipo di model utilizzato, come nell'Esempio 6.9.

Esempio 6.9

```
@model HomeViewModel

@{
   ViewBag.Title = "TypedAction";
}

<h2>@Model.Title</h2>
```

Il vantaggio di questo approccio è che ci ritroviamo a disposizione una proprietà `Model` che, in virtù di questa direttiva, è di tipo `HomeViewModel`, e quindi ci permette di accedere direttamente alle proprietà; in questo modo Visual Studio è in grado di accorgersi preventivamente se abbiamo sfruttato proprietà non valide o di fornirci supporto al momento della scrittura del codice tramite l'Intellisense, come possiamo vedere nella Figura 6.4.

Figura 6.4 – Intellisense durante la scrittura di una view.

ASP.NET Core MVC supporta una modalità di pre-compilazione delle view in fase di pubblicazione che può aiutarci a non commettere errori nelle view (ad esempio sbagliando a specificare il nome di una proprietà) e che si può attivare o disattivare agendo sul nodo `MvcRazorCompileOnPublish` all'interno del file `.csproj` (di default è attivata).

L'Esempio 6.10 mostra come attivare la funzionalità, che ha bisogno che sia referenziato il package `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` (solo se il target è .NET Framework, per .NET Core non è necessario), oppure ai metapackage `Microsoft.AspNetCore.All` (2.0) e `Microsoft.AspNetCore.All` (2.1).

Esempio 6.10

```
<Project Sdk="Microsoft.NET.Sdk.Web">
   <PropertyGroup>
      <TargetFramework>netcoreapp2.0</TargetFramework>
      <MvcRazorCompileOnPublish>true</MvcRazorCompileOnPublish>
   </PropertyGroup>
…
</Project>
```

Questa impostazione ha l'effetto collaterale di rendere leggermente più lenta la compilazione, soprattutto se abbiamo molte view nel progetto, ma ha il vantaggio di segnalare già a compile time eventuali incongruenze, come viene mostrato nella Figura 6.5.

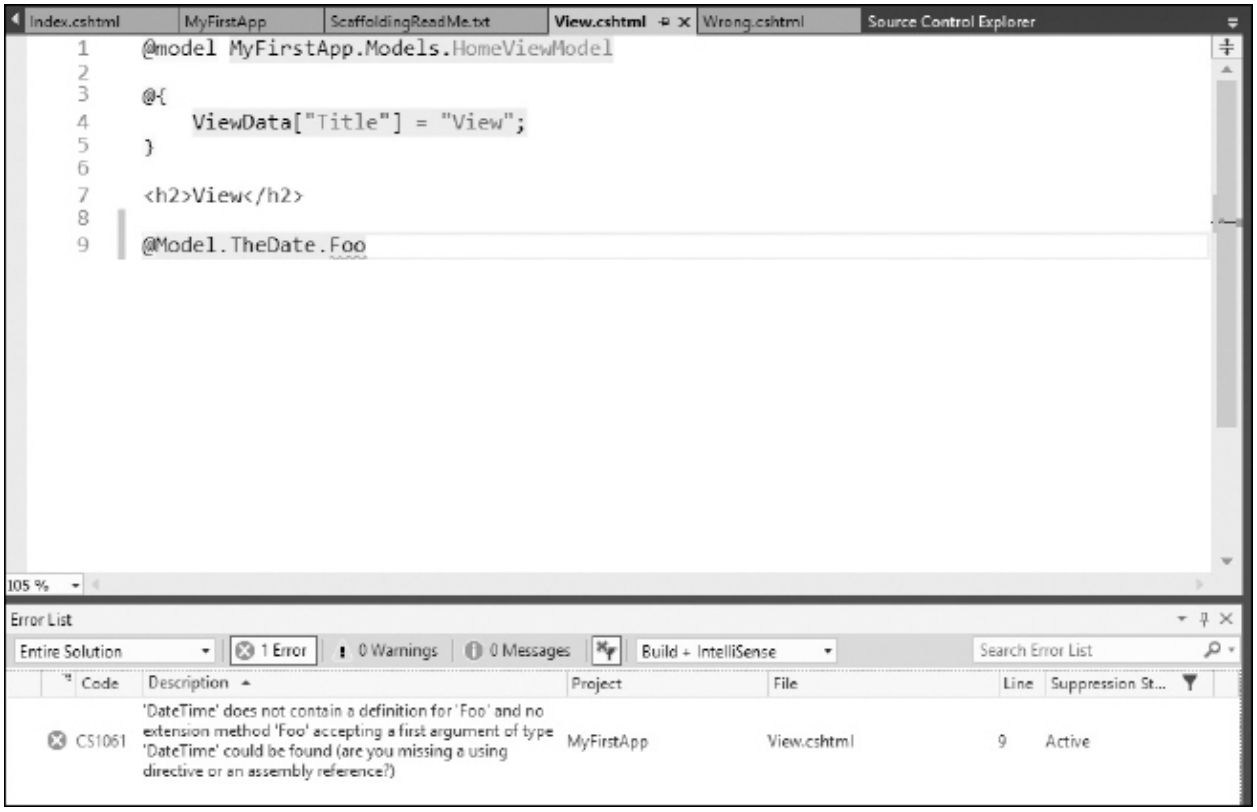


Figura 6.5 – Errore in compilazione di una view.

Grazie a questa funzionalità ASP.NET Core MVC provvederà a pre-compilare le view e, a differenza di ASP.NET MVC, le stesse non dovranno essere compilate da Razor in C# in fase di esecuzione, con benefici anche in termini di semplicità di deployment (non ci sono tanti file da distribuire) e cold start (l'avvio dell'applicazione a freddo), poiché i tempi si riducono e le performance migliorano nettamente.

Le view, per default, sfruttano i namespace specificati all'interno del file `_ViewImports.cshtml`, posto all'interno della directory `Views` del progetto, a differenza di ASP.NET MVC, che lo fa all'interno del `web.config` (che qui manca). Se abbiamo bisogno di utilizzarne altri ancora, possiamo modificare questo file, o importarli sistematicamente all'interno di ciascuna view, tramite la direttiva `@using`. L'Esempio 6.11 mostra entrambe queste soluzioni (la differenza è solo il file in cui vengono effettuate). Di default il template di progetto importa il namespace dell'applicazione e quello del suo model.

Esempio 6.11

```
@using MyApplication
@using MyApplication.Models
```

Con ASP.NET Core 2.1 è stata anche aggiunta la possibilità di distribuire una class library con all'interno blocchi di UI compilata a partire da file Razor. Avremo modo di tornare su questa funzionalità nel Capitolo 15.

A questo punto abbiamo sicuramente capito come l'uso di view fortemente tipizzate semplifichi di molto tutto il processo di sviluppo e abbia un peso fondamentale nell'abilità di accorgersi di eventuali problemi, prima che questi generino eccezioni. Si tratta di un requisito fondamentale per le applicazioni reali, ma di certo non è il solo. Una lacuna che dobbiamo colmare, per esempio, è la modalità secondo cui possiamo assicurare al nostro sito una consistenza grafica tra le varie pagine. Sarà l'argomento della prossima sessione.

Consistenza grafica tra le pagine: la layout view

Le layout page rappresentano uno strumento assolutamente indispensabile per mantenere uniforme il look delle pagine del nostro sito web. Tramite le layout page, infatti, possiamo definire un layout di base e un insieme di elementi di base, unitamente a una serie di placeholder che, nelle singole pagine interne, possiamo popolare con il contenuto specifico della pagina che l'utente sta visualizzando.

Le layout view sono delle normali view e pertanto non c'è una particolare tipologia di file da utilizzare: per aggiungerne una al nostro progetto non dobbiamo far altro che creare una nuova view, tipicamente all'interno della directory `Shared`, visto che con ogni probabilità dovrà essere accessibile da molteplici controller.

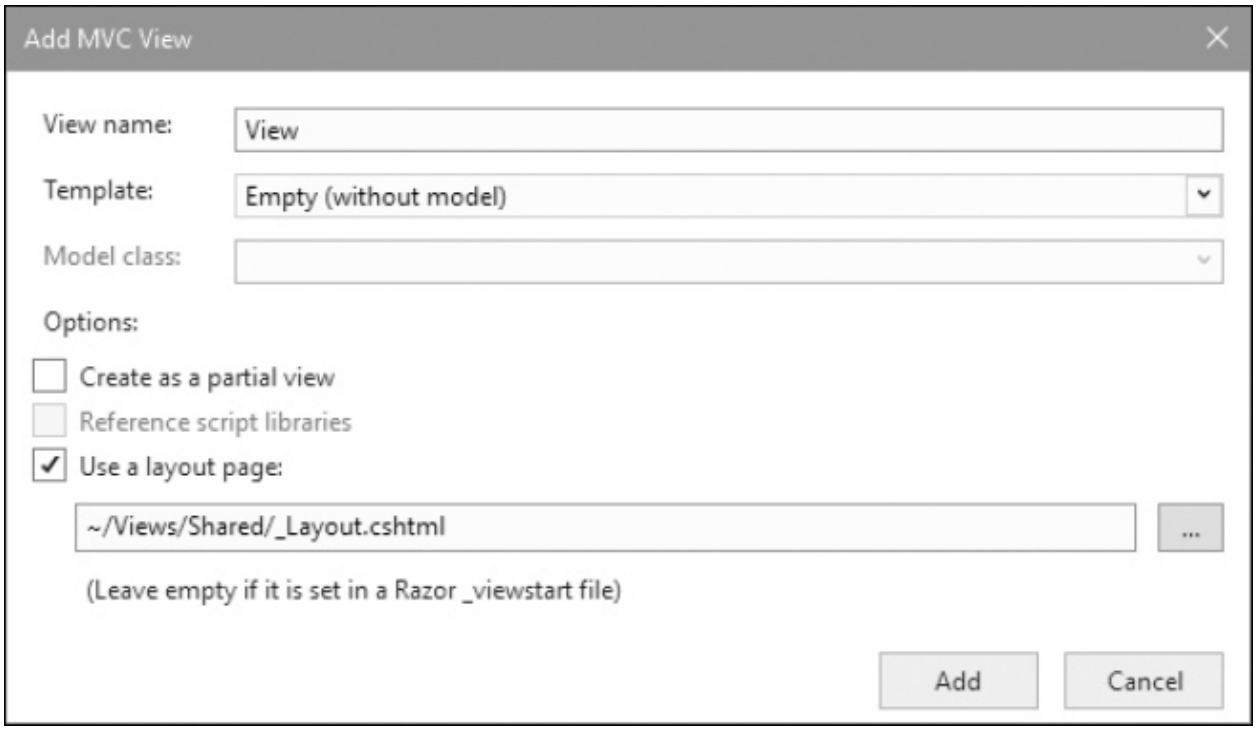


Figura 6.6 – Aggiunta di una layout view.

Come possiamo notare nella Figura 6.6, ci sono un paio di regole non scritte che vengono di solito adottate quando si crea una view di questo tipo.

- Il nome utilizzato è `_Layout.cshtml`; per convenzione, infatti, in ASP.NET Core MVC si preferisce utilizzare il carattere underscore come prefisso di tutte le view condivise.

- Non è una view tipizzata, perché sarà utilizzata in un gran numero di situazioni e vogliamo lasciare alle singole action la flessibilità di utilizzare il model più consono alla view che dovrà essere mostrata.

Sfruttare le layout view nel progetto

Dal punto di vista del codice, invece, non c'è nulla di nuovo rispetto alle regole viste finora, se non per la presenza, al suo interno, della chiamata al metodo `RenderBody`. L'Esempio 6.12 ci mostra una semplicissima layout view.

Esempio 6.12

```
<!DOCTYPE html>
<html>
   <head>
      ...
   </head>
   <body>
      <div id="body">
         @RenderBody()
      </div>
   </body>
</html>
```

Questo metodo rappresenta il segnaposto in cui, effettivamente, verrà renderizzato il contenuto specifico della pagina richiesta: ogni view ha una proprietà `Layout` che serve allo stesso scopo, e che dobbiamo valorizzare con l'URL della layout view desiderata.

Esempio 6.13

```
@model HomeViewModel
@{
   Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Contenuto specifico della pagina</h2>
```

Una cosa che dobbiamo dire è che questa proprietà può essere liberamente valorizzata anche a runtime, prima della fase di execute della view.

> *ASP.NET Core MVC supporta anche layout view innestate; come abbiamo avuto modo di specificare, una layout view è a tutti gli effetti una normalissima view e pertanto nulla ci vieta di impostarne la proprietà Layout in modo che punti a un'ulteriore layout view.*

Generalmente, si opta per una registrazione globale della layout view, come vedremo nel prossimo paragrafo di questo capitolo.

La view _ViewStart

Impostare la proprietà `Layout` per ogni view della nostra applicazione può essere sicuramente tedioso e poco semplice da manutenere, nel momento in cui, in futuro, dovessimo decidere di modificare questi riferimenti. Per questa ragione, ASP.NET Core MVC mette a disposizione uno strumento, ossia una view speciale, denominata `_ViewStart`.

Si tratta di un file di Razor, all'interno del quale possiamo definire del codice, e che ha la caratteristica di essere eseguito, in maniera del tutto automatica, prima del rendering di una qualsiasi view. Questo ci consente di specificare all'interno di `_ViewStart` quale sarà la layout view che sarà adottata per default da tutte le pagine dell'applicazione, senza che dobbiamo valorizzarla all'interno di ogni file, come nell'Esempio 6.14.

Esempio 6.14

```
@{
   Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Se diamo una nuova occhiata alla Figura 6.1, presente all'inizio di questo capitolo, possiamo notare che, tipicamente, il file `_ViewStart` viene creato direttamente all'interno della directory *Views*. In realtà, in un progetto possono coesistere molteplici file `_ViewStart`, ognuno in una specifica directory. Il risultato è che il runtime li eseguirà in sequenza, partendo da quello più esterno fino a quello più specifico, contenuto nella directory della view selezionata. Nella Figura 6.7, per esempio, abbiamo creato un file `_ViewStart` all'interno della directory `Home`.

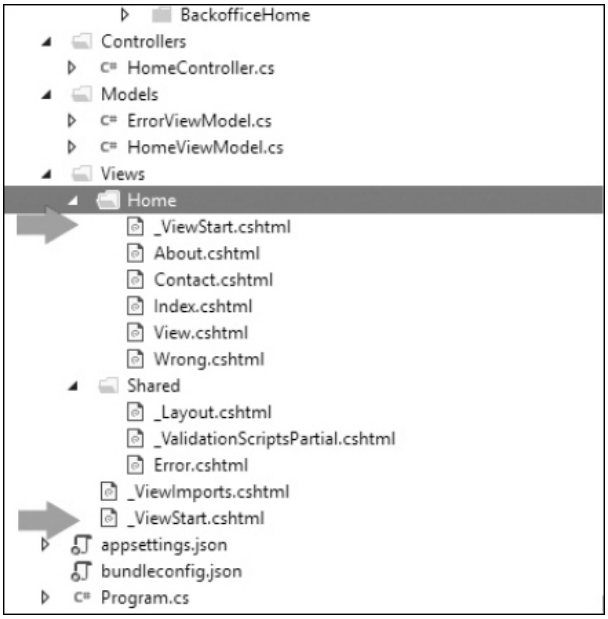


Figura 6.7 – Una struttura di directory con _ViewStart dentro home.

Se a questo punto proviamo a invocare l'action `Index`, verranno eseguiti, nell'ordine:

- Il file `_ViewStart.cshtml` contenuto nella directory `Views`.
- Il file `_ViewStart.cshtml` contenuto nella directory `Home`.

Questo ci permette, per esempio, di specificare una view di layout differenti per aree funzionali del sito, o addirittura sul singolo controller.

Com'è lecito attendersi, in ogni caso, l'ultima parola spetta sempre alla view che verrà renderizzata, che a sua volta potrà indicare una layout view specifica.

Definire sezioni aggiuntive in una layout view

Oltre al contenuto di default, che viene incluso tramite il metodo `RenderBody`, una content view può definire anche delle sezioni aggiuntive, ognuna identificata da un nome, tramite la direttiva `@section`, come nell'Esempio 6.15.

Esempio 6.15

```
@section Footer {
   <p>Footer della content view</p>
}
```

Questi blocchi vengono ignorati da `RenderBody`, ma richiedono un metodo `RenderSection` all'interno della layout view, per specificare dove devono essere posizionati.

Esempio 6.16

```
<div id="body">
   @RenderBody()
</div>

<div id="footer">
   @RenderSection("Footer")
   @RenderSection("OptionalSection", required:false)
</div>
```

Per default, ogni sezione aggiuntiva deve essere presente nella content view, altrimenti verrà sollevato un errore a runtime, a meno che non specifichiamo il contrario con il parametro `required`, come abbiamo fatto nell'Esempio 6.16 per *OptionalSection*.

L'alternativa è sfruttare il metodo `IsSectionDefined` per scoprire se una sezione è definita nella content view, come nell'Esempio 6.17.

Esempio 6.17

```
<div id="footer">
   @if (IsSectionDefined("Footer"))
   {
      @RenderSection("Footer")
   }
   else
   {
      <p>Contenuto di default definito su layout view</p>
   }
   @RenderSection("OptionalSection", required: false)
</div>
```

Questa tecnica, come possiamo vedere nel codice precedente, può essere utilizzata anche per produrre un contenuto di default.

Le layout page, insomma, sono uno strumento estremamente potente, grazie al quale possiamo realizzare un sito web che si visualizzi correttamente su ogni dispositivo utilizzato, abbinandolo all'uso di file CSS che facciano uso di tecniche come il responsive design, che di default nel template predefinito viene attuato grazie all'utilizzo di Bootstrap. Al momento, però, siamo ancora costretti a scrivere manualmente la totalità dell'HTML della pagina. Uno strumento messo a disposizione da ASP.NET Core MVC per rendere più semplice la scrittura del codice di una view è rappresentato dagli HTML e dai tag helper. Ne parleremo nella prossima sezione.

Semplificare il codice delle view: gli HTML e i tag helper

Se abbiamo già lavorato con ASP.NET Web Forms, sicuramente uno degli aspetti che, al primo impatto, sembrano più ostici nell'approccio ad ASP.NET Core MVC è l'assenza di controlli server side evoluti: quando realizziamo le view, infatti, siamo costretti a "sporcarci le mani" con il vecchio codice HTML, mentre in realtà ci piacerebbe lavorare a un livello più alto di astrazione dei semplici tag.

Questi tipi di necessità sono gestite in ASP.NET Core MVC tramite gli HTML helper, ossia dei metodi che possiamo sfruttare per produrre in maniera automatica dei blocchi di markup, anche complessi. Mutuati da ASP.NET MVC, gli HTML Helper in ASP.NET Core MVC

sono affiancati anche dai Tag helper, che ne semplificano ulteriormente le potenzialità e che ricordano più da vicino i controlli di ASP.NET Web Forms.

Analizzeremo prima gli HTML helper e poi passeremo, invece, ad analizzare i Tag helper.

Gli HTML Helper

Dal punto di vista strettamente pratico, si tratta di extension method della classe `HtmlHelper`, che è esposta dalla view tramite la proprietà `Html`. Come possiamo notare nella Figura 6.8, ASP.NET Core MVC definisce diversi metodi di questo tipo, per la maggior parte sfruttabili per la creazione di elementi utili per le form, quali `TextBox`, `ListBox` o `RadioButton`, solo per citarne alcuni.

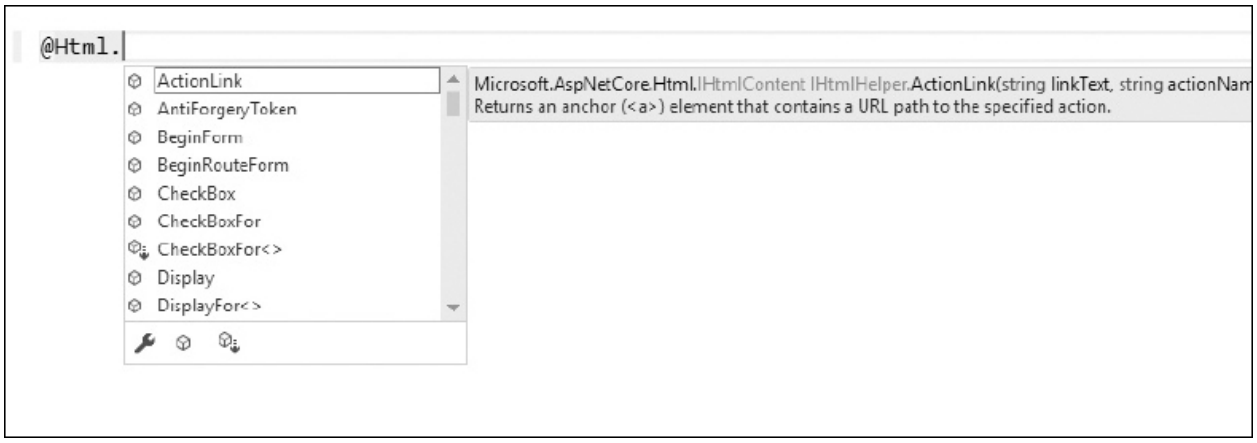


Figura 6.8 – HTML helper standard di ASP.NET Core MVC.

Daremo una rapida occhiata a questi argomenti, per poi spostarci subito ai Tag helper, che sono molto più interessanti.

Il metodo ActionLink

Quando dobbiamo inserire un link verso un'altra pagina del nostro sito, l'utilizzo diretto di un URL scritto staticamente in un tag `<a>` non è consigliabile, perché ci vincola a specificare l'URL di destinazione in maniera prefissata nel codice della view mentre, come abbiamo visto finora, in generale questo è il prodotto della configurazione del routing.

Una strada più semplice è sfruttare l'HTML helper `ActionLink`, tramite il quale possiamo specificare la destinazione nei termini di controller e action. L'Esempio 6.18 mostra alcuni casi di utilizzo.

Esempio 6.18

```
@* Link alla action SomeAction dello stesso controller *@
@Html.ActionLink("Semplice", "SomeAction")

@* Link alla action Index di CustomersController *@
@* In virtù dei default, l'URL sarà /Customers *@
@Html.ActionLink("Con controller", "Index", "Customers")

@* genera <a href=".." target="_blank"> *@
@Html.ActionLink("Attributi personalizzati", "Index",
   "Customers", null, new {
      target = "_blank", style = "font-size:20px" })
@* Genera un link all'URL /Customers/Detail/23 con la classe css
    myLink *@
@Html.ActionLink("Link con parametri", "Detail", "Customers",
new { id = 23 }, new { @class = "myLink" })
```

Come possiamo notare nel codice precedente, esistono diversi overload che possiamo sfruttare per generare varie tipologie di URL e personalizzare il markup generato. In particolare, per indicare i parametri addizionali o attributi HTML, abbiamo utilizzato un anonymous type; esiste anche un overload che invece sfrutta un `RouteValueDictionary`, ma la sintassi dell'esempio è sicuramente più comoda.

Questo metodo, oltre al vantaggio di rendere indubbiamente più semplice la determinazione dell'URL, ha anche la peculiarità di adattarsi alla configurazione del routing: se in futuro dovessimo decidere di modificare queste impostazioni, infatti, la consistenza dei link del nostro sito web rimarrebbe comunque assicurata.

In alcune occasioni, per esempio se stiamo scrivendo del codice JavaScript, potremmo avere la necessità di recuperare solo l'URL di una determinata route, piuttosto che generare un vero e proprio link. In questo caso possiamo sfruttare la classe `UrlHelper` come nell'Esempio 6.19.

Esempio 6.19

```
<script type="text/javascript">
   function myFunction() {
      var url =
         '@Url.Action("Detail", "Customers", new { id = 23 })';
      window.open(url);
   }
</script>
```

Il risultato del metodo `Action` è una stringa contenente l'URL desiderato, che nel codice precedente abbiamo usato per popolare la variabile `url`.

Il metodo RouteLink

L'helper `ActionLink` che abbiamo visto nella sezione precedente è sicuramente molto versatile, ma ha il limite di funzionare esclusivamente con route di ASP.NET Core MVC. Nell'ambito di applicazioni complesse, che magari sfruttano molteplici tecnologie contemporaneamente, non è detto che siano definite solo route di questo tipo. Nel Capitolo 4, infatti, abbiamo visto che in generale una route può definire diversi parametri e gestire la chiamata con un qualsiasi `IRouteHandler`; il fatto di avere parametri nella route quali controller e action, insomma, è solo un caso particolare, per quanto comune.

Se abbiamo bisogno di lavorare più a basso livello, possiamo sfruttare l'HTML helper `RouteLink`. Immaginiamo di aver definito, nella classe `RouteConfig`, la route dell'Esempio 6.20.

Esempio 6.20

```
routes.Add("CustomRoute",
   new Route("Custom/{first}/{second}/{third}",
new CustomRouteHandler()));
```

`ActionLink` non è in grado di generare un link a questa route, perché non contiene nozioni di controller o action. In questo caso, allora, dobbiamo sfruttare `RouteLink` come nell'Esempio 6.21, che ci permette di specificare la route e indicare puntualmente i `routeValues` desiderati per determinare l'URL.

Esempio 6.21

```
@*Genera un link a /Custom/First/Second/Third*@

@Html.RouteLink("Link a customRoute", "customRoute",
new { first = "First", second = "Second", third = "Third" })
```

Anche in questo caso, se invece di costruire un link dobbiamo semplicemente determinare l'URL, possiamo sfruttare il metodo `Url.RouteUrl`.

Il metodo Raw

Da ciò che abbiamo appreso finora, per includere in pagina un contenuto variabile tutto ciò che dobbiamo fare è esporre il dato tramite una proprietà del model e referenziarlo all'interno del markup, come nell'Esempio 6.22.

Esempio 6.22

```
<p>@Model.SomeProperty</p>
```

In linea generale, visualizzare in pagina contenuto variabile, che magari proviene dal database o da un precedente input dell'utente, è un'operazione che comporta un certo grado di rischio, visto che potremmo essere esposti ad attacchi di tipo XSS, che approfondiremo nel Capitolo 17. In realtà, finora non abbiamo mai messo in luce l'argomento perché, per ovvie ragioni di sicurezza, Razor effettua automaticamente l'encoding del testo.

Esistono dei casi, tuttavia, in cui vogliamo disabilitare l'encoding, magari perché `SomeProperty` contiene del testo formattato, che altrimenti non verrebbe visualizzato correttamente. In questi casi possiamo usare l'helper `Raw` dell'Esempio 6.23.

Esempio 6.23

```
<p>@Html.Raw(Model.SomeProperty)</p>
```

Per le ragioni che abbiamo citato, però, dobbiamo essere molto attenti quando usiamo questo metodo e prendere le dovute precauzioni, magari rimuovendo da `SomeProperty` eventuali tag potenzialmente dannosi, come `<script>`, `<iframe>` e via discorrendo. In generale, se vogliamo visualizzare una stringa a video senza che Razor ne faccia l'encoding, sarà sufficiente utilizzare il tipo `HtmlString`, in luogo di `string`.

Il metodo Partial e le partial view

Le layout page che abbiamo introdotto nel corso di questo capitolo svolgono sicuramente una funzione fondamentale nell'ottica di mantenere costante il look del nostro sito web tra le diverse pagine. Purtroppo, però, da sole non sono sufficienti a risolvere il problema

nella sua interezza. Spesso, infatti, ci troviamo nella necessità di replicare più volte, in diverse view, alcuni blocchi di contenuto; per i casi semplici abbiamo a disposizione gli HTML helper, ma quando il markup diviene complesso e le variabili in gioco sono molteplici, abbiamo bisogno di uno strumento più versatile: le partial view.

Per questa tipologia di view, valgono le stesse regole che abbiamo illustrato fino a questo momento, a partire dalla modalità di creazione, che avviene tramite la solita finestra di dialogo della Figura 6.9, che abbiamo visto più volte nel corso di questo capitolo. L'unica differenza rispetto ai casi precedenti è rappresentata dalla spunta sulla voce *Create as partial view*.

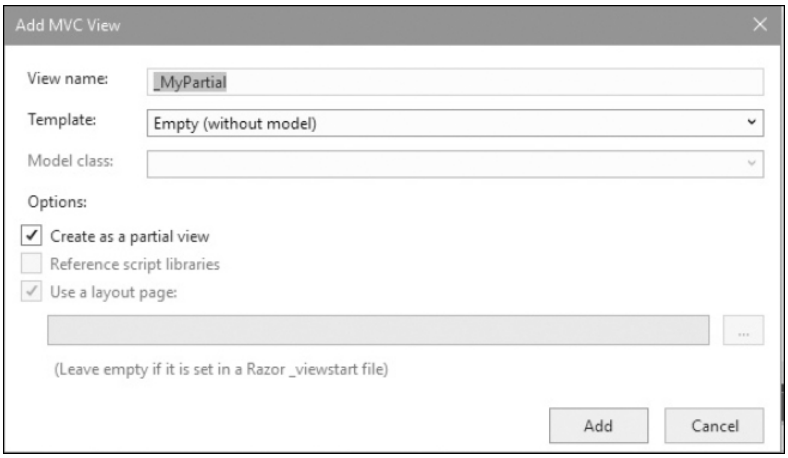


Figura 6.9 – Creazione di una partial view.

Come possiamo notare nella figura, le partial view possono essere tipizzate o non tipizzate e, ovviamente, non hanno mai un riferimento a una layout view, visto che non sono della pagine autonome, ma vanno inserite all'interno di una content view. Dal punto di vista del codice, non c'è nulla di nuovo rispetto a quanto abbiamo visto finora, come possiamo notare nell'Esempio 6.24, relativo a una partial view tipizzata.

Esempio 6.24

```
@model HomeViewModel

<p>Questo paragrafo appartiene alla partial view</p>

<p>@Model.Title</p>
```

Per visualizzare una partial view all'interno della pagina, dobbiamo sfruttare l'HTML helper `Partial`, come nell'Esempio 6.25.

Esempio 6.25

```
@Html.Partial("MyPartialView", Model)
```

Questo metodo accetta come parametro principale il nome della view; nel caso del codice precedente, ASP.NET cercherà un file di nome `MyPartialView.cshtml` all'interno della directory del controller in esecuzione, o in Views\Shared, dove dobbiamo posizionare il file nel caso in cui vogliamo che possa essere referenziato da diversi controller. Nell'Esempio 6.25, inoltre, abbiamo sfruttato un particolare overload per passare anche un'istanza del model, visto che la nostra partial view ne ha bisogno.

I Tag helper e la gestione del markup

Come abbiamo visto, lo scopo per cui sono stati introdotti gli HTML helper è quello di semplificare la creazione del markup, essendo di fatto delle classi particolari che si incastrano con l'infrastruttura di rendering di Razor.

Tutto quello che scriviamo è, infatti, preso da Razor e convertito in una classe C#, con chiamate che ricostruiscono il codice che abbiamo scritto e che, a runtime, genereranno l'HTML corrispondente.

Un concetto caro a chi ha sviluppato con ASP.NET Web Forms è quello dei componenti, cioè di markup che viene inserito nella pagina e poi viene programmato, evitando di ragionare in funzione di solo codice: quest'ultimo, infatti, è sempre soggetto ad errori ed è scarsamente apprezzato da web designer o sviluppatori front-end che lavorano ad una view, a cui bisogna insegnare specificatamente come agire. Per questo motivo, sono nati i Tag helper, che rappresentano pezzi di markup inseriti nella pagina e che poi a runtime produrranno un output specifico, esattamente come gli HTML Helper.

Di fatto, rappresentano l'anello di giunzione tra HTML helper e HTML e non vanno a sostituirsi ai primi: Tag helper e HTML helper convivono serenamente. Non è detto che ci sia un Tag helper per ogni HTML helper e viceversa. Ci sono casi in cui vale la pena adottare una scelta e casi in cui è meglio affidarsi a un'altra. Avremo modo di analizzare i principali Tag helper subito e rimandare agli altri nel prossimo capitolo, perché, come vedremo, sono l'ideale proprio in presenza di form.

Il Tag helper Partial

Un tag helper minimale, introdotto con ASP.NET 2.1, può essere quello dell'Esempio 6.26, che consente di effettuare il rendering di una partial view.

Esempio 6.26

```
<partial name="_MyPartialView" asp-for="Model" />
```

Come si può notare, la chiamata precedente lascia posto ad un tag HTML, che poi Razor trasformerà nello stesso risultato, cioè quello di includere la view parziale. In questo caso, grazie all'attributo `name` possiamo specificare la view a cui fare riferimento, mentre con `asp-for` andiamo a passare il model. Tecnicamente, quello che facciamo con quest'ultima proprietà è specificare una `ModelExpression`, che poi andrà ad assegnare il modello, come nel caso dell'HTML helper. Con le stesso approccio è possibile passare anche un ViewData specifico, agendo sulla proprietà `view-data` del tag.

Rispetto all'utilizzo della sintassi basata su HTML helper, in questo caso il rendering è sempre effettuato in maniera asincrona.

Il Tag helper Environment

Un caso particolarmente interessante tra i Tag helper predefiniti è quello denominato `Environment`. Nei capitoli precedenti, in particolare nel Capitolo 3, abbiamo visto come ASP.NET Core supporti già il concetto di ambiente di esecuzione. Questo tag helper non fa altro che rendere possibile la differenziazione del markup emesso in funzione dell'ambiente e ritorna molto comodo in fase di sviluppo, per caricare file JavaScript o CSS non minified, lasciando questa eventualità solo alla produzione, piuttosto che per emettere markup specifico, evitando del tutto di affidarsi a un blocco di codice con una condizione, che potrebbe più facilmente contenere errori.

Nell'Esempio 6.27 possiamo vedere come sfruttare questo Tag helper.

Esempio 6.27

```
<environment include="Staging,Production">
   <strong>Staging o Production</strong>
</environment>
<environment exclude="Staging">
   <strong>Production o Development</strong>
</environment>
```

Nell'esempio possiamo notare due utilizzi, legati agli attributi `include` e `exclude`, che servono rispettivamente a specificare (separati da virgola) gli ambienti per cui emettere quel markup, oppure quelli da escludere (sempre separati da virgola).

Non c'è molto altro da aggiungere, poiché, in caso la condizione non dovesse venire rispettata, nessun markup verrebbe emesso dal Tag helper.

Il Tag helper Anchor

Uno dei primi esempi di HTML helper che abbiamo introdotto è stato quello per generare link attraverso `ActionLink`. Questo rappresenta una delle cause per cui è più facile che uno sviluppatore alle prime armi commetta degli errori, poiché l'unico modo di passare i valori a questo metodo è per posizione. Il grande vantaggio dei Tag helper, invece, è quello di essere molto espressivi.

L'Esempio 6.18, prima introdotto, può essere "tradotto" con i tag riportati nell'esempio che segue.

Esempio 6.28

```
@* Link alla action SomeAction dello stesso controller *@

<a asp-action="SomeAction">Semplice</a>

@* Link alla action Index di CustomersController
      In virtù dei default, l'URL sarà /Customers *@
<a asp-controller="Customers"
   asp-action="Index">Con controller</a>
@* Genera <a href=".." target="_blank"> *@
<a asp-controller="Customers"
   asp-action="Index"
   target="_blank"
   style="font-size:20px">Attributi personalizzati</a>
@* Genera un link all'URL /Customers/Detail/23 con la classe css myLink *@
<a asp-controller="Customers"
   asp-action="Index"
   asp-route-id="23"
class="myLink">Attributi personalizzati</a>
```

Per uno sviluppatore HTML (o che non ha mai visto ASP.NET MVC in precedenza) questa sintassi risulta molto più leggibile e gestibile. In realtà, è semplice da utilizzare per chiunque, poiché ci sono poche proprietà che regolano il modo in cui verrà generato l'URL finale.

- `asp-controller`: indica il controller da utilizzare e, se omesso, presuppone lo stesso controller abbinato alla action che ha renderizzato la view;
- `asp-action`: per indicare la action da invocare;
- `asp-route-{parametro}`: per indicare il parametro di routing da passare.

Tutti gli altri attributi, se non riconosciuti, verranno lasciati come sono e, pertanto, renderizzati nell'HTML risultante.

Analogamente, è possibile referenziare anche una route per nome, specificando solo l'attributo `asp-route`, o passare tutti i parametri di routing (sotto forma di `Dictionary`) grazie all'attributo asp-all-route-data, come si vede nell'Esempio 6.29.

Esempio 6.29

```
@* corrisponde ad una route che porta a /Profile/ *@
<a asp-route="MyProfile">Il mio profilo</a>
@* Passa tutti i parametri in un colpo solo alla route *@
@{
var parms = new Dictionary<string, string>
   {
      { "searchKey", "ASP.NET" },
      { "author", "Daniele Bochicchio" }
    };
 }

 <a asp-route="SearchForBooks"
      asp-all-route-data="parms">Tutti i libri di Daniele Bochicchio
   su ASP.NET</a>
```

Per concludere, esiste un attributo `asp-fragment` che consente di passare un fragment all'URL, cioè quella parte che si trova nell'URL dopo il carattere # e che serve a navigare internamente ad un documento HTML.

Il Tag helper Image

Per concludere questa prima carrellata, analizziamo il caso del Tag helper Image, che serve per generare un'immagine HTML a cui viene aggiunto un numero di versione, così da non avere problemi con la cache lato client del browser. Nell'Esempio 6.30 possiamo trovare un esempio di come attivare il tag.

Esempio 6.30

```
<img src="~/images/brand.png"
asp-append-version="true" />
```

L'output generato sarà molto simile a quello dell'Esempio 6.30.

Esempio 6.31

```
<img src=" /images/brand.png?v=APzJduEowspUX8NrqJLwjZrRj3SflDUbO3UjD9MFgy0" />
```

Il valore assegnato al parametro `v` è quello dell'hash Sha512 del file su disco, che viene calcolato e tenuto in cache, per essere invalidato e calcolato di nuovo ad ogni modifica del file. Il risultato è che il browser, ricevendo un URL differente, provvederà a scaricare nuovamente il file, in caso di cambiamento.

Conclusioni

In questo capitolo abbiamo chiuso il cerchio sul pattern model-view-controller, entrando nel dettaglio sulle modalità con cui, in ASP.NET Core MVC, possiamo realizzare le view, ossia i componenti ultimi che servono a produrre l'effettivo markup HTML che verrà visualizzato sul browser utente.

Una view è un file che integra al suo interno markup e codice C# e che viene processato da un view engine che prende il nome di Razor. Si tratta, a tutti gli effetti, di un "linguaggio" inedito, per cui inizialmente ci siamo soffermati sulla sua sintassi di base.

Successivamente, abbiamo illustrato gli strumenti che abbiamo a disposizione in questa tecnologia: grazie alle layout view, siamo in grado di realizzare dei template comuni a tutte le pagine, così che il nostro sito web appaia consistente dal punto di vista grafico. Gli HTML helper, invece, offrono un supporto differente, orientato alla generazione del markup: è il caso di ActionLink e RouteLink, grazie ai quali possiamo creare link alle pagine in base alle impostazioni di routing, o di Partial che, rispettivamente, sfruttano il concetto di partial view per componentizzare porzioni di interfaccia, così che siano riutilizzabili in molteplici pagine. Inoltre, abbiamo anche visto come con i Tag helper possiamo sfruttare una nuova modalità, maggiormente orientata all'inserimento di tag HTML speciali nella pagina.

Tuttavia, a questo punto, siamo ancora a metà del nostro percorso di apprendimento. Infatti non siamo ancora in grado di gestire correttamente l'interazione utente e di accettare l'input di dati. Si tratta di un argomento piuttosto vasto, che affronteremo nel prossimo capitolo, dove daremo ancora maggior risalto, come anticipato, all'utilizzo dei Tag helper.

7
Gestione delle form

Arrivati a questo punto del libro abbiamo capito come funziona ASP.NET Core MVC e come i suoi tre principali componenti (il Model, la View e il Controller) interagiscono tra loro per permetterci di scrivere codice più facilmente testabile e riutilizzabile. In questo capitolo cominciamo a entrare nel vivo di ASP.NET Core MVC, mostrando come gestire le form.

La gestione di una form richiede molta attenzione in quanto ci sono diversi punti che devono essere presi in considerazione. Innanzitutto c'è la gestione dei campi di input, ossia come generarne il codice HTML, come mostrare la loro label, e così via: si tratta di un'operazione che è di molto semplificata grazie alle data annotation e agli appositi helper di ASP.NET Core MVC, che in combinazione ci permettono di generare una form in tempi brevissimi e con pochissimo codice.

Una volta mostrata la form all'utente, dobbiamo prestare molta attenzione a un altro punto importante come la validazione dei dati. Così come per la visualizzazione della form, anche la validazione dei dati è un compito che possiamo assolvere utilizzando le data annotation e gli specifici HTML helper senza fare altro.

In questo capitolo analizzeremo queste tre fasi, vedendo anche come personalizzare alcuni aspetti come il codice per generare la form e la personalizzazione del sistema di validazione per validare i tipi di dati non previsti di default (codice fiscale, partita IVA e così via). Cominciamo ora col vedere come generare una form.

Generare la form da un model

La form che andremo a creare è una semplice form che gestisce i dati di un cliente. Per ovvi scopi dimostrativi, questa nostra form contiene solo alcuni dati che ci permettono di mostrare in maniera esaustiva le funzionalità messe a disposizione da ASP.NET Core MVC.

Di fatto, ogni form all'interno di un controller è implementata attraverso due action: una si occuperà di mostrare la form all'utente, l'altra sarà invocata quando l'utente invierà la form stessa dal client al server. Questo comportamento è abbastanza standard in tutte le applicazioni web.

Una volta che l'utente ha inserito i dati validi e questi sono stati inviati al server, nella action che gestisce la richiesta dobbiamo gestire i dati per poterli poi elaborare. Recuperare i dati è un compito che non svolge la action bensì il model binder, il quale si preoccupa di recuperare i dati della richiesta e mapparli sui parametri della action, in modo che questa si debba occupare solo di usufruire dei dati e non di andarli anche a recuperare. In particolare, il compito del model binder è quello di valorizzare questi parametri in base al contenuto della richiesta. In parole povere, se è presente un campo con lo stesso nome di una proprietà del model e un valore con questi compatibile, ci penserà il model binder a fare tutto il lavoro sporco: il model binder è molto potente ed è in grado di ricreare anche oggetti complessi, facendoci lavorare direttamente su un modello (come vedremo) e senza necessità di specificare direttamente ciascun campo.

Prima di addentraci maggiormente in questo ambito, però, dobbiamo cominciare definendo il model.

Definire il model

La classe che definisce il cliente contiene un id, un nome, l'indirizzo, lo stato di residenza, la data di registrazione e un booleano che identifica se l'utente è attivo.

Dello stato di residenza gestiamo l'id e non il nome in quanto nella form vogliamo usare una dropdown dalla quale l'utente seleziona lo stato di residenza. Questo significa che dobbiamo avere nel model una lista di oggetti che rappresentano gli stati.

Una volta scritte le classi, dobbiamo cominciare a decorare le proprietà con le data annotation, che altro non sono che una serie di attributi che il motore di ASP.NET Core MVC interpreta per eseguire delle azioni. Queste annotation sono le stesse di .NET Framework, per cui i prossimi contenuti possono essere affrontati con una certa velocità da chi ha già dimestichezza con questa modalità.

Vediamo in dettaglio quali sono le data annotation che servono per i nostri scopi.

L'attributo Display

L'attributo `Display` permette di specificare la label da associare alla proprietà. La proprietà più importante di `Display` è `Name` tramite la quale specifichiamo la label. Se decidiamo di localizzare la nostra applicazione, possiamo utilizzare la proprietà `ResourceType`, alla quale passiamo il tipo del file di risorse mentre nella proprietà `Name` impostiamo il nome della chiave nel file. L'Esempio 7.1 mostra come utilizzare quest'attributo su una proprietà.

Esempio 7.1 – C#

```
//Stringa normale
[Display(Name="Indirizzo")]
public string Address { get; set; }

//Stringa da file di risorse
[Display(Name="Indirizzo", ResourceType=typeof(Labels))]
public string Address { get; set; }
```

Come vedremo più avanti, l'attributo `Display` viene interpretato dagli helper di ASP.NET Core MVC per renderizzare la label associata al campo input, che rappresenta la proprietà. Vediamo ora un altro attributo fondamentale, comodo per personalizzare il contenuto del campo che mostra la proprietà.

L'attributo DisplayFormat

L'attributo `DisplayFormat` permette di personalizzare come il valore di una proprietà deve essere visualizzato. Per fare un esempio, nella classe che rappresenta il cliente abbiamo una proprietà che rappresenta la data. Per default, quando deve essere renderizzata una data, ASP.NET Core MVC visualizza il risultato della chiamata al metodo `ToString`, il quale restituisce la data con inclusa l'ora e questo non è sempre quello che vogliamo. Grazie all'attributo `DisplayFormat` possiamo intervenire nel processo di valutazione della data e decidere il formato da applicare.

La proprietà principale dell'attributo `DisplayFormat` è `DataFormatString`, che permette di specificare la formattazione da applicare al valore della proprietà marcata con l'attributo. La seconda proprietà fondamentale è `ApplyFormatInEditMode` tramite la quale specifichiamo se la formattazione deve essere applicata sia se stiamo editando sia se stiamo solo visualizzando la proprietà (il valore di default è `false`, il che significa che la formattazione viene applicata solo quando il campo deve essere visualizzato e non quando deve essere modificato). L'ultima proprietà importante dell'attributo `DisplayText` è `NullDisplayText`, tramite la quale possiamo specificare cosa visualizzare se la proprietà è null. L'Esempio 7.2 mostra come utilizzare `DisplayFormat`.

Esempio 7.2

```
[DisplayFormat(DataFormatString = "{0:d}",
   ApplyFormatInEditMode = True)]
public DateTime RegistrationDate { get; set; }
```

Come si vede nel codice, utilizzare l'attributo `DisplayFormat` è molto semplice e quindi non necessita di ulteriori spiegazioni. Nella prossima sezione ci occupiamo dell'attributo `UIHint`, che ci permette di specificare il template da utilizzare per la proprietà.

L'attributo UIHint

L'attributo `UIHint` permette di specificare il template che ASP.NET Core MVC deve utilizzare per mostrare la proprietà sia quando questa deve essere visualizzata sia quando deve essere modificata.

Il concetto di template non è stato ancora introdotto ma ne parleremo approfonditamente nel prosieguo del capitolo.

La proprietà più importante (e spesso la sola utilizzata) di `UIHint` è `UIHint` tramite la quale impostiamo il nome del template da utilizzare per la proprietà, così come viene mostrato nell'Esempio 7.3.

Esempio 7.3

```
[UIHint("Address")]
public string Address { get; set; }
```

Grazie agli attributi `Display`, `DisplayFormat` e `UIHint` possiamo descrivere le caratteristiche delle singole proprietà del model. Sfruttando questi metadati, tramite gli HTML e i tag helper possiamo creare le view che renderizzano il model in maniera molto semplice. Nel prossimo esempio possiamo vedere il codice completo del model che implementeremo in questo capitolo.

Esempio 7.4

```
public class CustomerModel
{
   public CustomerModel()
   {
      Countries = new List<CountryModel>();
      Contacts = new List<ContactModel>();
   }
   public int Id { get; set; }
   [Display(Name = "Nome")]
   public string Name { get; set; }
   [DisplayFormat(DataFormatString = "{0:d}",
      ApplyFormatInEditMode=true)]
    [Display(Name = "Data registrazione")]
    public DateTime RegistrationDate { get; set; }
    [DisplayFormat(DataFormatString = "{0:n2}",
       ApplyFormatInEditMode = true)]
    [Display(Name = "Sconto")]
    public decimal DiscountPercentage { get; set; }
    [Display(Name = "Attivo")]
    public bool IsActive { get; set; }
```

```
    [Display(Name = ”Indirizzo”)]
    public string Address { get; set; }
    [Display(Name = ”Stato”)]
    public int CountryId { get; set; }
    public List<CountryModel> Countries { get; set; }
 }

 public class CountryModel
 {
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Adesso che il nostro modello è pronto, possiamo creare il controller con la action che gestisce la richiesta di creazione del cliente.

Definire il controller e la action

Per definire il controller dobbiamo creare la classe `CustomersController` e aggiungere il metodo `Create` che funge da action. Oltre a creare la action che restituisce la view tramite la quale l'utente inserisce i dati, dobbiamo anche creare una view che gestisce i dati quando l'utente li rinvia al server. Il codice del controller e della action è visibile nell'Esempio 7.5.

Esempio 7.5

```
public class CustomersController : Controller
{
   public IActionResult Create()
   {
      var model = new CustomerModel();
      model.Countries.AddRange(GetCountries()); //recupera gli stati
      return View(model);
   }
   [HttpPost]
   public IActionResult Create(CustomerModel model)
   {
      ...
   }
}
```

Il codice del metodo che gestisce i dati di ritorno dal client (quello con l'attributo `HttpPost` in testa) è volutamente vuoto, in quanto verrà approfondito nelle sezioni successive. Per ora è importante prendere in considerazione solamente la sua firma, che accetta in input un oggetto `CustomerModel` che, come abbiamo anticipato, verrà creato dal model binder.

L'ultimo passo per creare la form è la costruzione della view.

Creare la view

La creazione del model e del controller è stato un passo che non ha mostrato nulla di nuovo rispetto a quanto abbiamo appreso nei capitoli precedenti. Ora che cominciamo a parlare della view iniziamo invece a sfruttare nuove funzionalità, la prima delle quali è l'utilizzo degli helper per la creazione del codice HTML.

Benché continuino a essere supportati, gli HTML helper relativi alla form (tra cui troviamo `BeginForm`) vengono sempre meno utilizzati, in favore dei corrispondenti tag helper.

Per questo motivo, inizieremo a dare un'occhiata ai nuovi tag helper relativi alle form e sorvoleremo, invece, sui corrispondenti HTML helper.

Il tag helper Form

Il primo tag helper da affrontare è quello relativo alle form, che consente di inviare il contenuto delle stesse in seguito all'azione dell'utente sul pulsante associato. In particolare, nell'Esempio 7.6 possiamo notare che il modello a oggetti di questo tag helper ricalca molto da vicino quello utilizzato dal tag helper Anchor, introdotto alla fine del precedente capitolo.

Esempio 7.6

```
<form method=”post”
   asp-controller=”Customers”
   asp-action=”Create”>
   … resto della form
   <input type=”submit” value=”Invia dati” />
</form>
```

Eventuali parametri di routing possono essere specificati mediante l'attributo `asp-route-*`, già introdotto nel capitolo precedente e con `asp-route` si può fare riferimento a una route per nome. In realtà, i parametri `asp-controller` e `asp-action`, in questo caso, possono essere omessi, poiché la form, per comportamento predefinito, farà post allo stesso URL da cui viene eseguita. Sono stati specificati per far comprendere come poter utilizzare questo helper in scenari differenti, quando c'è da puntare a una form che invierà i dati a una action di un controller differente, per esempio per una form di ricerca messa all'interno della layout page.

In realtà il tag helper Form fa molto di più, generando anche un token di validazione della richiesta, su cui torneremo in maggior dettaglio nel Capitolo 18, dedicato agli aspetti di sicurezza. Un modello del markup generale è disponibile nell'esempio che segue.

Esempio 7.7

```
<form method="post" action="/Customers/Create">
   … resto della form e dei pulsanti
   <input name="__RequestVerificationToken" type="hidden" value="…rimosso…" />
</form>
```

Ora che abbiamo visto come generare la form, passiamo a vedere come generare il campo di testo che permette di inserire il nome del cliente.

Il tag helper Input

L'utente deve inserire il nome del cliente attraverso una textbox.

Per generare una textbox, possiamo utilizzare gli HTML helper `TextBox` e `TextBoxFor`, i quali restituiscono in output un oggetto `MvcHtmlString` contenente un tag `input` con l'attributo `type` impostato su `text`. Il primo parametro di entrambi i metodi rappresenta la proprietà del model che vogliamo associare alla textbox. La differenza tra i metodi consiste nel fatto che la proprietà viene espressa tramite una stringa nel caso del metodo `TextBox` e tramite una lambda expression nel caso del metodo `TextBoxFor`.

> *Il metodo TextBoxFor offre una sintassi tipizzata, quindi è da preferire al metodo TextBox. Tuttavia, non esistono solo TextBox e TextBoxFor; tutti gli HTML helper che generano campi di input hanno una versione tipizzata e una non tipizzata (come vedremo nel corso del capitolo) e quella tipizzata è da preferire sempre.*

In realtà, l'equivalente con i tag helper è quello di utilizzare il tag helper Input, che si fa all'omonimo tag dell'HTML. Nell'Esempio 7.8 possiamo vedere come utilizzare entrambi i metodi.

Esempio 7.8

```
@Html.TextBoxFor(m => m.Name)
@Html.TextBox("Name")
<input type="text" asp-for="Name" />
```

In questo caso, l'uso di un approccio o dell'altro potrebbe sembrare simile, anche se appare evidente che l'uso dei tag helper semplifichi notevolmente la vita a chi preferisce sviluppare (giustamente) direttamente in HTML.

La differenza è tutta visibile, come già abbiamo visto con la generazione di link, quando dobbiamo passare argomenti aggiuntivi. Per loro stessa natura, infatti, i tag helper emettono nel markup generato tutti gli attributi che non riconoscono direttamente, con una sintassi molto semplice.

Nell'Esempio 7.9 possiamo vedere come generare una textbox, a cui assegniamo la classe CSS `big`.

Esempio 7.9

```
@Html.TextBoxFor(m => m.Name, new { @class = "big" })
@Html.TextBox("Name", new { @class = "big" })
<input type="text" asp-for="Name" class="big" />
```

Per questo motivo, non parleremo più specificamente di HTML helper, ma ci soffermeremo solo sui tag helper, che sono stati creati appositamente per semplificare la vita agli sviluppatori, senza perdere feature come l'Intellisense, a cui siamo abituati. Quello che viene specificato dopo l'attributo `asp-for`, infatti, viene recuperato direttamente dal tipo del modello associato, garantendo un'ottima esperienza in fase di sviluppo, anche per un eventuale web designer che dovesse partecipare al progetto, e un ulteriore controllo a compile time.

Tecnicamente, il valore dell'attributo `asp-for` è di tipo `ModelExpression`, per cui quello che viene inserito nell'attributo diventerà la parte destra di una lambda, dalla forma `m => m.Name`, nel nostro caso. Quando ASP.NET Core MVC cerca il valore da assegnare, lo andrà a recuperare prima da una chiave del `ModelState` il cui nome è lo stesso specificato e poi dal risultato dell'espressione `Model.Name`, dove `Name` è il nome della proprietà specificata. In caso di proprietà aggiuntive, basterà specificarle nella consueta forma, per esempio come `Address.Country`.

Un altro dato che l'utente deve inserire è il flag che identifica se il cliente è attivo. Nella prossima sezione vediamo come permettere all'utente di inserire questo dato.

Corrispondenza tra tipi e tag Helper

Il modo migliore per far inserire all'utente una proprietà booleana è mostrare nell'interfaccia una checkbox. Nell'Esempio 7.10 possiamo vedere l'utilizzo del solito tag helper Input in questi scenari.

Esempio 7.10

```
<input type="text" asp-for="IsActive" />
```

In questo caso, potremmo omettere o usare l'attributo type, che verrà automaticamente generato in base al modello.

Il modo in cui il tag helper Input decide che tipo di HTML produrre è influenzato dal tipo di dato o dalla data annotation utilizzata. La Tabella 7.1 chiarisce meglio i meccanismi alla base di questo sistema, il cui obiettivo è di generare tag HTML5, pronti anche per il mobile.

Tabella 7.1 – Corrispondenza tra tipi, data annotation e HTML prodotto.

Tipo o [data annotation]	Valore della proprietà type
String	text
Bool	checkbox
DateTime	datetime
[DataType(DataType.Date)]	date
[DataType(DataType.Time)]	time
Byte Int Single Double	number
[HiddenInput]	hidden
[Url]	url
[DataType(DataType.Password)]	password
[Email]	email

Come si può intuire, quindi, nel nostro caso, avendo già usato opportunamente le data annotation, il compito è molto semplificato.

Per capire meglio come funzioni il meccanismo alla base di questo tag helper, facciamo un esempio concreto. Una delle necessità più comuni sul web è quella di avere la necessità di memorizzare nella pagina un dato senza doverlo però rendere visibile sulla pagina stessa. Nel nostro caso, l'id del cliente è un candidato perfetto. I campi hidden assolvono proprio questo compito, in quanto immagazzinano un dato ma non lo mostrano a video. Per generare un campo hidden, possiamo utilizzare un codice come quello mostrato nell'Esempio 7.11.

Esempio 7.11

```
<input type="hidden" asp-for="Id" />
```

In questo caso, non avendo specificato la data annotation `[HiddenInput]` sulla proprietà `Id`, ci saremmo aspettati di non poter esplicitare il tipo di campo: in realtà, come abbiamo visto nell'esempio precedente, il tag helper Input consente di gestire comodamente questo aspetto. A vincere sarà sempre quello che viene specificato dallo sviluppatore all'interno delle proprietà.

Come abbiamo detto all'inizio di questa sezione, l'utente deve inserire lo stato di residenza sfruttando una dropdown e non inserendo il nome dello stato. Nella prossima sezione vedremo come creare la dropdown.

Il tag helper Select

Attraverso questo tag helper possiamo creare una dropdown e specificare quale elemento deve essere selezionato. L'attributo `asp-for` conterrà la proprietà del model che rappresenta l'id dell'elemento selezionato e quella `asp-items` la lista degli elementi che popolano la dropdown, che è una lista di oggetti di tipo `SelectListItem`. Nell'Esempio 7.12 possiamo vedere come usare questo tag helper.

Esempio 7.12

```
@{
   var countries = Model.Countries.Select(c =>
      new SelectListItem() {
         Text = c.Name,
         Value = c.Id.ToString()
   });
}

<select asp-for="CountryID" asp-items="@countries">
```

```
   <option value="">Seleziona un valore...</option>
 </select>
```

Molto spesso capita di voler inserire un elemento vuoto in una dropdown, perché la selezione di un elemento non è obbligatoria o perché vogliamo mettere un messaggio personalizzato come primo elemento (per esempio, "seleziona un elemento"). In questo caso, dobbiamo semplicemente agire sull'HTML prodotto, mettendo come primo valore quello che vogliamo anteporre al contenuto degli effettivi items, che saranno poi aggiunti.

> *Nell'Esempio 7.12 abbiamo trasformato una lista di oggetti CountryModel in una lista di oggetti SelectListItem. Volendo, possiamo anche modificare il model per trasformare la proprietà Countries da List<CountryModel> a List<SelectListItem> così da non dover effettuare la trasformazione nella view. La scelta dell'uso di una tecnica piuttosto che di un'altra è strettamente personale, in quanto il risultato che si ottiene è il medesimo. Questo controllo consente anche di specificare un gruppo (attraverso la proprietà Group).*

La dropdown è un controllo di input molto utile quando si ha a che fare con molti elementi.

Questo Helper genererà in automatico l'attributo `multiple="multiple"` nell'HTML, qualora la proprietà specificata all'interno di `asp-for` sia di tipo `IEnumerable`, come, per esempio, quando vogliamo consentire una scelta multipla agli utenti rispetto a un elenco di valori consentiti.

Il tag helper Textarea

Analogamente a quanto offerto da Input, il tag helper `Textarea` consente di generare il corrispondente tag HTML, utile per gestire input su più linee.

Come si può vedere nell'Esempio 7.13, può utilizzare le data annotation `[MinLength]` e `[MaxLength]` per controllare in automatico la relativa grandezza del campo e, come vedremo più avanti nel capitolo, anche la relativa validazione.

Esempio 7.13 – C#

```
 public class Customer
 {
    … altre proprietà
    [MinLength(10)]
    [MaxLength(1000)]
 public string Notes { get; set; }
```

Esempio 7.13

```
<textarea asp-for="Notes"></textarea>
```

> *Lo stesso effetto può essere ottenuto utilizzando la data annotation [StringLength], nella forma [StringLength(maximumLength: 1000, MinimumLength = 10)].*

Finora abbiamo sfruttato i tag helper che creano i campi di input per l'utente.

Facciamo un passo indietro e torniamo a dare un'occhiata agli HTML helper, perché ci tornano utili per uno scenario particolare: visualizzare a video, formattati già a dovere, i dati contenuti all'interno di un campo.

Tutti i metodi visti finora si sono concentrati sul mostrare o modificare il valore di una proprietà. Il prossimo metodo, invece, si occupa di renderizzare una label da associare ai campi di input.

Il tag helper Label

Il tag helper Label renderizza un tag `label` associato al campo relativo alla proprietà che i metodi accettano in input. Il valore della label viene preso dall'attributo `Display` presente sulla proprietà. Se l'attributo `Display` non è presente, allora viene utilizzato il nome della proprietà. Nell'Esempio 7.14 possiamo vedere il codice necessario.

Esempio 7.14

```
<label asp-for="Name"></label>
```

L'uso di questo tag helper è molto semplice, ma è molto indicato, in quanto migliora l'usabilità delle form in presenza di browser non visuali, come quelli testuali e vocali, aggiungendo gli attributi per l'accessibilità.

Ognuno degli approcci che abbiamo visto finora ha la caratteristica di corrispondere a un particolare tag HTML e pertanto questi metodi specificano in maniera puntuale quale tipo di input vogliamo utilizzare per una determinata proprietà. Il prossimo metodo che andiamo ad analizzare offre la possibilità di non legare la proprietà a un input specifico, sfruttando invece il concetto di template.

Gli HTML helper Editor e EditorFor

I metodi `Editor` e `EditorFor` sono metodi che renderizzano il campo di input relativo a una proprietà in base a un template. Il template da utilizzare viene deciso direttamente dal runtime di ASP.NET Core MVC, in base al tipo della proprietà da renderizzare. Per

esempio, se vogliamo creare il campo di input di una proprietà booleana, ASP.NET Core MVC sceglie il template che mostra una checkbox; se, invece, vogliamo creare il campo di input di un numero, una stringa o una data, ASP.NET Core MVC sceglie il template che mostra una textbox.

Grazie a `Editor` e `EditorFor` non dobbiamo preoccuparci di usare i metodi appositi, ma possiamo usare un metodo solo e lasciare che sia ASP.NET Core MVC a renderizzare il campo di input corretto.

> *I template di default sono contenuti all'interno degli assembly di ASP.NET Core MVC. Tuttavia possiamo personalizzare i template come viene mostrato nel prosieguo di questa sezione.*

Come per tutti gli altri metodi, `Editor` e `EditorFor` accettano come primo parametro il nome di una proprietà, come è possibile vedere nell'Esempio 7.15.

Esempio 7.15

```
@Html.EditorFor(m => m.Name)

@Html.Editor("Name")
```

Se il template di default di ASP.NET Core MVC non fosse sufficiente a soddisfare le nostre esigenze, possiamo personalizzare il template creando semplicemente un file `nometipo.cshtml` nella directory `Views/Shared/EditorTemplates` (la directory dove MVC cerca i template per generare i campi che permettono di editare una proprietà, detta anche directory degli editor template). Per esempio, se vogliamo personalizzare il template che genera il campo di input per una data, dobbiamo creare il file `DateTime.cshtml` e inserire il codice nel file. Nell'Esempio 7.16 possiamo vedere il codice del template.

Esempio 7.16

```
@model DateTime

@Html.TextBox("", Model, new { @class = "date" })
```

Come si vede nell'esempio che abbiamo appena preso in esame, all'interno del template viene usato il metodo `TextBox` e non `TextBoxFor`, in quanto il model della partial view non è il model della pagina chiamante, bensì il tipo della proprietà che il template deve gestire che, in questo caso, è `DateTime`.

Se la proprietà ha un attributo `UIHint`, ASP.NET Core MVC non usa il template specifico per il tipo della proprietà, bensì sfrutta nella directory degli editor template la partial view con il nome specificato nell'attributo `UIHint`.

Nel caso in cui vogliamo avere più editor per lo stesso tipo (per esempio, perché vogliamo un campo di testo diverso a seconda che si tratti l'indirizzo o il C.A.P), possiamo utilizzare un overload del metodo `Editor` o `EditorFor`, che accetta come secondo parametro il nome del template da utilizzare (e che ASP.NET Core MVC cercherà comunque nella directory degli editor template) e, come opzione, i dati da passare al template.

L'ultimo metodo che analizziamo è molto simile a quello che abbiamo appena visto, con la differenza che si occupa di visualizzare la proprietà in sola visualizzazione e non di permetterne la modifica.

Gli HTML helper Display e DisplayFor

Questo è uno dei casi in cui un HTML Helper non è stato affiancato da un corrispondente tag helper, poiché i tag helper, come il nome stesso suggerisce, servono per mappare 1:1 il corrispondente tag HTML e l'obiettivo, in questi casi, è in realtà solo quello di mostrare un valore già formattato, ma senza aggiungere tag HTML intorno, che restano a cura dello sviluppatore.

I metodi `Display` e `DisplayFor` sono metodi che renderizzano in base a un template il valore di una proprietà. Il template da utilizzare viene deciso direttamente dal runtime di ASP.NET Core MVC in base al tipo della proprietà da renderizzare. Per esempio, se vogliamo mostrare il valore di una proprietà booleana, ASP.NET Core MVC sceglie il template che mostra una checkbox disabilitata; se vogliamo mostrare il valore di una proprietà numerica, una stringa o una data, ASP.NET Core MVC sceglie il template che mostra il valore formattato secondo le regole specificate dall'attributo `DisplayFormat` o, se non è presente l'attributo, secondo la culture del thread corrente.

Come per tutti gli altri metodi, `Display` e `DisplayFor` accettano come primo parametro il nome di una proprietà, come risulta visibile nell'Esempio 7.17.

Esempio 7.17

```
@Html.DisplayFor(m => m.Name)

@Html.Display("Name")
```

Tutte le regole di selezione del template e gli overload visti per i metodi `Editor` e `EditorFor` valgono in egual modo per `Display` e `DisplayFor` e quindi non c'è la necessità di specificarli nuovamente in questa sezione.

Arrivati a questo punto abbiamo in mano tutti gli strumenti per creare la view per il nostro model. Mostrare l'intero codice della view sarebbe eccessivamente lungo e poco utile, in quanto i vari esempi presentati nel corso di questa sezione, una volta messi insieme, formano il codice completo della view e che è disponibile negli allegati di questo libro.

Nella Figura 7.1 possiamo vedere la view così come è renderizzata dal browser.

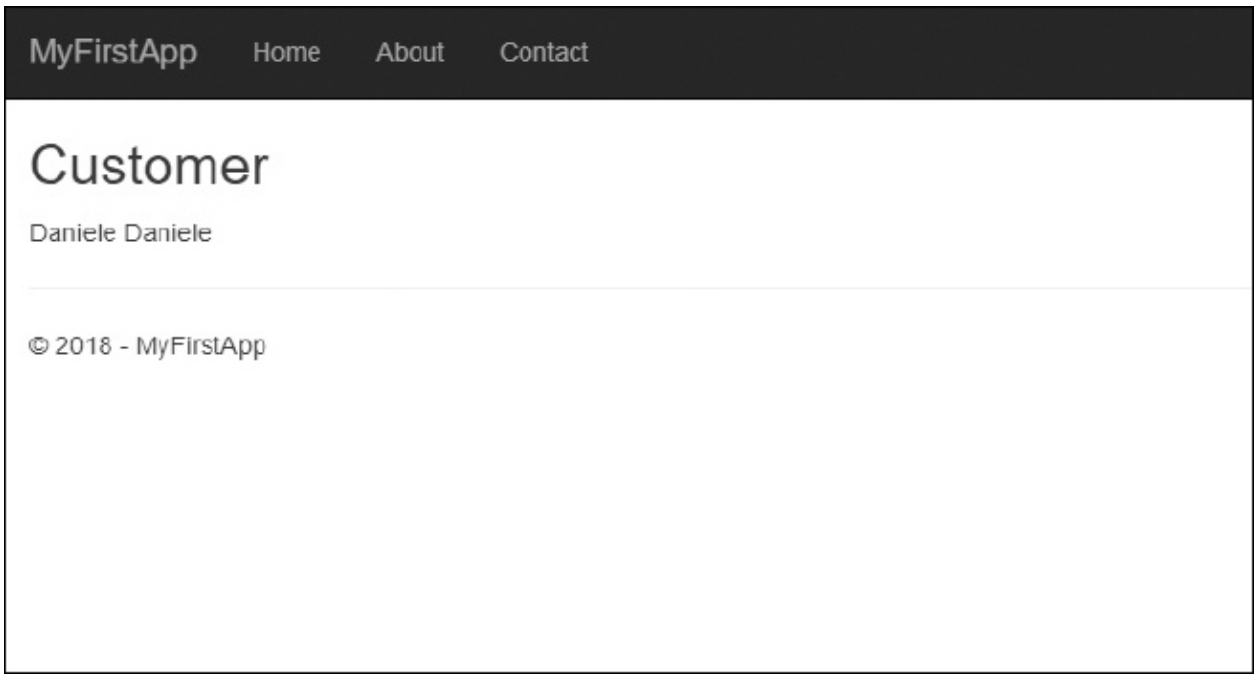


Figura 7.1 – La view per editare i dati del model.

Ora che abbiamo finito di parlare degli HTML helper che generano i controlli sulla form, cambiamo argomento e vediamo come validare i dati inseriti dall'utente in questi controlli.

Validare l'input dell'utente

La regola principale di qualunque applicazione è quella di non fidarsi dei dati inseriti dall'utente. ASP.NET Core MVC offre un framework di validazione molto ampio e personalizzabile in qualunque punto, così da permetterci di validare ogni tipo di dato sia lato client sia lato server.

Il framework di ASP.NET Core MVC si basa sulle data annotation. Il motore di ASP.NET Core MVC interpreta gli attributi di validazione (che vedremo nel corso di questa sezione) ed effettua due passaggi per attivare la verifica sull'input dell'utente.

Il primo passo si verifica quando vengono eseguiti gli HTML e i tag helper che abbiamo visto nella sezione precedente. Questi metodi, in base alle data annotation, emettono codice HTML o JavaScript (a seconda che la unobtrusive validation, tecnica di cui parleremo più avanti, sia abilitata o meno) che viene successivamente interpretato dalle librerie JavaScript sul client per eseguire la validazione dell'input.

Il secondo passo avviene nel momento in cui i dati vengono inviati al server. La action che viene eseguita è quella che accetta in input un oggetto `CustomerModel` (Esempio 7.5). Prima che la action venga eseguita, entra in azione il model binder che ricostruisce l'oggetto `CustomerModel` sfruttando i dati provenienti dal client. Durante la ricostruzione, il model binder esegue la validazione della classe. Il risultato è tale che, quando viene eseguita la nostra action, la validazione è già stata effettuata e tutto quello che dobbiamo fare è controllare che non ci siano stati errori e comportarci di conseguenza.

Grazie a questa infrastruttura, la validazione dei dati è perfettamente integrata nel sistema e noi dobbiamo compiere pochissimi passi per abilitarla. Inoltre, inserire la nostra validazione personalizzata (per esempio, per validare un codice fiscale) è estremamente banale, in quanto ci basta creare un attributo di validazione e applicarlo sulle proprietà e quindi scrivere il codice JavaScript necessario a eseguire la validazione lato client.

Come abbiamo detto in precedenza, lato client, ASP.NET Core MVC si basa su una libreria che interpreta il codice emesso dagli HTML helper: questa libreria è il plugin `jquery.validate`. Cominciamo ora col vedere gli attributi di validazione utilizzati da ASP.NET Core MVC.

Gli attributi di validazione

Gli attributi di validazione all'interno delle data annotation utilizzati da ASP.NET Core MVC sono in tutto cinque: `Required`, `Range`, `RegularExpression`, `StringLength` e `Remote`. Nel corso di questa sezione li esamineremo tutti in dettaglio così da poterli usare al meglio. Cominciamo però non da uno degli attributi elencati, bensì dalla classe base di tutti gli attributi di validazione.

La classe ValidationAttribute

La classe `ValidationAttribute` è la classe base di tutti gli attributi di validazione. Oltre a contenere un po' di logica legata alla validazione, questa classe espone le tre proprietà che ci permettono di impostare i messaggi di errore:

- `ErrorMessage`: imposta il messaggio di errore;
- `ErrorMessageResourceType`: tipo della classe legata a un file di risorse (nel caso in cui vogliamo localizzare la nostra applicazione);
- `ErrorMessageResourceName`: nome della chiave nel file di risorse specificato nella proprietà `ErrorMessageResourceType`.

Facendo parte della classe base da cui tutti gli attributi ereditano, queste proprietà sono esposte da tutti gli attributi di validazione. Adesso che abbiamo visto come personalizzare i messaggi di errore di tutti, passiamo ad analizzare l'attributo `Required`.

L'attributo Required

L'attributo `Required` specifica che una proprietà è obbligatoria. Questo attributo è necessario solo per le proprietà che possono assumere un valore nullo (cioè quelle proprietà che espongono un tipo per riferimento come `string`). Le proprietà che non possono essere nulle (quelle che espongono un tipo per valore, come `Int32`, `Decimal`, `Boolean` e così via) sono considerate automaticamente come obbligatorie; l'Esempio 7.18 mostra l'utilizzo dell'attributo `Required` sulla proprietà `Name` del nostro model.

Esempio 7.18

```
[Required]
public string Name { get; set; }
```

Il secondo attributo che andiamo ad analizzare è quello che permette di validare un range di dati.

L'attributo Range

L'attributo `Range` permette di specificare il limite minimo e massimo di un valore. Nel nostro modello abbiamo la proprietà `DiscountPercentage` che, essendo una percentuale, è una perfetta candidata per essere validata tramite l'attributo `Range`.

Le proprietà principali di `Range` sono `MinimumValue` e `MaximumValue` che contengono rispettivamente il valore minimo e il valore massimo che la proprietà può assumere e che possono essere impostate direttamente tramite costruttore così come mostra l'Esempio 7.19.

Esempio 7.19

```
[Range(0, 100)]
public decimal DiscountPercentage { get; set; }
```

Un altro attributo molto importante è quello che valida i dati in base a una regular expression.

L'attributo RegularExpression

L'attributo `RegularExpression` permette di specificare una regular expression in base alla quale validare la proprietà. Nel nostro model abbiamo la proprietà `Name` che deve essere validata con una espressione in quanto, rappresentando il nome di una persona, non può contenere numeri ma solo caratteri alfabetici, spazi e apici (questo vale per la lingua italiana ma, volendo, possiamo costruire espressioni più complesse per altre lingue). La proprietà tramite la quale esprimiamo la regular expression è `Pattern` e può essere impostata direttamente tramite costruttore, come viene mostrato nell'Esempio 7.20.

Esempio 7.20

```
[RegularExpression("[A-Za-z 'àéèìòù]+")]
public string Name { get; set; }
```

Le regular expression sono molto potenti, ma possono risultare molto complesse da apprendere. Nella prossima sezione vediamo un attributo che semplifica la validazione delle stringhe in un determinato caso, evitando così l'uso delle regular expression.

L'attributo StringLength

L'attributo `StringLength` permette di impostare la lunghezza minima (opzionale) e massima (obbligatoria) di una stringa. Se supponiamo che nel nostro model non siano ammessi nomi più lunghi di 40 caratteri, possiamo utilizzare `StringLength` per specificare questa regola.

Le proprietà che esprimono la lunghezza minima e massima sono rispettivamente `MinimumLength` e `MaximumLength`. La prima deve essere specificata usando la sintassi di inizializzazione della proprietà, mentre la seconda può essere impostata direttamente tramite il costruttore, come viene mostrato nel prossimo esempio.

Esempio 7.21

```
[StringLength(40, MinimumLength = 1)]
public string Name { get; set; }
```

L'ultimo attributo di cui parliamo è quello che abilita la validazione remota.

L'attributo Remote

L'attributo `Remote` permette di specificare una action da eseguire per validare i dati della proprietà e che viene invocata sia dal client, prima di inviare i dati al server, sia dal server, quando valida i dati.

Per specificare la action da eseguire, dobbiamo sfruttare il costruttore che accetta in input il nome della action e del controller. La action deve accettare in input il valore da validare e restituire in output un oggetto `Bool` all'interno di una risposta JSON, contenente `true` o `false` a seconda che il valore sia valido o no. L'Esempio 7.22 mostra l'utilizzo dell'attributo, mentre nell'Esempio 7.23 è riportato il codice della action di validazione.

Esempio 7.22

```
public class Customer
{
   // … altre proprietà
   [Remote(action: "IsNameValid", controller: "Customers")]
   public string Name { get; set; }
}
```

Esempio 7.23

```
[AcceptVerbs("Get", "Post")]
public IActionResult IsNameValid(string Name)
{
   var isValid = false;
   //Logica di validazione
   return Json(isValid);
}
```

Gran parte del lavoro è fatto da ASP.NET Core MVC, che lo invia al server con una richiesta AJAX e mostra il risultato della validazione, eventualmente bloccando il submit della form se la validazione non è andata a buon fine.

Torneremo sulla validazione lato client tra un attimo, nel prosieguo di questo capitolo.

Altri tipi di validazione

Per completare il discorso, occorre menzionare anche queste data annotation, che vengono automaticamente utilizzate nelle View per generare delle validazioni:

- `CreditCard`: per validare formalmente un numero di carta di credito.
- `Compare`: compara due proprietà tra di loro, utile per confermare password o indirizzi e-mail.
- `EmailAddress`: per validare formalmente un indirizzo e-mail.
- `Phone`: per validare formalmente un numero di telefono.
- `Url`: per validare formalmente un URL.

Con questo chiudiamo la carrellata degli attributi di validazione e cambiamo argomento, passando a vedere come sfruttare questi attributi sulla view.

Applicare la validazione sulla view

Nella sezione precedente abbiamo detto che gli attributi di validazione vengono interpretati dagli HTML e dai tag helper, i quali generano il codice HTML o JavaScript che viene poi interpretato dal plugin `jquery.validate` per effettuare la validazione sul client. Questo significa che, sul client, tutto ciò che dobbiamo fare è referenziare i file JavaScript e specificare come vogliamo mostrare i messaggi di errore. Fortunatamente il primo passo è estremamente semplice in quanto le librerie da referenziare sono già incluse nel template di progetto di ASP.NET Core MVC e fanno riferimento alla AJAX CDN DI Microsoft. Il secondo passo consiste nell'utilizzare alcuni helper creati appositamente per la validazione. Nell'Esempio 7.24 (tra un paio di paragrafi) si può notare un pezzo di codice da includere (per esempio, direttamente nella layout page, piuttosto che nelle singole pagine) per fare riferimento a queste librerie.

Tuttavia, prima di cominciare a sfruttare la validazione sul client, parliamo di una caratteristica di validazione molto importante: l'unobtrusive validation.

Unobtrusive validation

Fino all'avvento di HTML5, tutti i framework di validazione JavaScript si affidavano alla definizione di regole di validazione attraverso dei metodi da invocare. Questo significa che, in pagine di grandi dimensioni con molti campi da validare, bisognava emettere molto codice JavaScript che rallentava la pagina.

Con l'introduzione di HTML5 e dell'attributo `data-`, i framework di validazione hanno cambiato approccio e adesso le regole di validazione non sono più espresse tramite codice JavaScript, bensì con degli attributi `data-` inseriti direttamente nel tag HTML che intendiamo validare.

In questo modo, generare il codice di validazione diventa estremamente semplice, in quanto gli HTML helper che renderizzano i campi di input non devono far altro che leggere gli attributi di validazione e generare i relativi attributi HTML. Inoltre, le pagine vengono caricate in maniera più rapida, in quanto non c'è tutto il codice JavaScript da scaricare, compilare ed eseguire.

L'unobtrusive validation è abilitata di default e consigliamo vivamente di farne uso in tutte le form, per migliorare notevolmente l'usabilità delle stesse: non c'è motivo di far aspettare l'utente che la risposta venga generata dal server, se un valore è già formalmente non valido.

Ora che abbiamo capito come le regole di validazione vengono trasferite dagli attributi server al client, cominciamo a vedere come lavorare con le view, partendo dalle referenze ai file JavaScript.

Referenziare le librerie JavaScript di validazione

I file JavaScript delle librerie di validazione sono presenti nella directory `Scripts` del progetto e possono essere referenziati direttamente importandoli nella nostra view. Tuttavia, il modo migliore per importarli è quello di referenziarli dalla AJAX CDN, così da essere sicuri che siano presenti in tutte le pagine senza dover fare nulla, così come illustrato nell'Esempio 7.24, che mostra il contenuto del file `_ValidationScriptsPartial.cshtml`.

Esempio 7.24

```
<environment include="Development">
    <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js">
</script>
</environment>
<environment exclude="Development">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.17.0/
    jquery.validate.min.js"
              asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
              asp-fallback-test="window.jQuery && window.jQuery.validator"
              crossorigin="anonymous"
              integrity="sha384-rZfj/ogBloos6wzLGpPkkOr/gpkBNLZ6b6yLy4o+ok+t/
              SAKlL5mvXLr0OXNi1Hp">
    </script>
    <script src="https://ajax.aspnetcdn.com/ajax/jquery.validation.unobtrusive/
              3.2.9/jquery.validate.unobtrusive.min.js"
              asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.
              unobtrusive.min.js"
              asp-fallback-test="window.jQuery && window.jQuery.validator &&
              window.jQuery.validator.unobtrusive"
              crossorigin="anonymous"
              integrity="sha384-ifv0TYDWxBHzvAk2Z0n8R434FL1Rlv/Av18DXE43N/1rvHyOG4iz
Kst0f2iSLdds">
    </script>
</environment>
```

A partire da ASP.NET Core 2.1, questi file sono automaticamente referenziati, all'interno di una Partial View, chiamata `_ValidationScriptsPartial.cshtml` e contenuta in `Shared`.

Gli helper di validazione

Gli HTML helper `ValidationMessage` e `ValidationMessageFor` hanno lo scopo di mostrare il messaggio di errore associato alla proprietà che gestiscono. Questi metodi non hanno alcun effetto sulla validazione in quanto devono solo mostrare il messaggio all'utente; questo significa che potremmo anche non usare questi metodi e la pagina validerebbe i dati lo stesso.

Lo stesso discorso si applica al tag helper Validation, che in questo caso agisce su un tag `span` attraverso l'attributo `asp-validation-for`.

Come per tutti gli helper, il primo parametro dei metodi rappresenta la proprietà di cui bisogna mostrare il messaggio di errore, come viene mostrato nell'Esempio 7.25.

Esempio 7.25 – Con HTML helper

```
@Html.LabelFor(m => m.Name)
@Html.EditorFor(m => m.Name)
@Html.ValidationMessageFor(m => m.Name)
```

Esempio 7.25 - Con tag helper

```
<div class="form-group">
      <label asp-for="Name" class="col-md-2 control-label"></label>
      <div class="col-md-10">
           <input asp-for="Name" class="form-control" />
           <span asp-validation-for="Name" class="text-danger"></span>
      </div>
</div>
```

In genere, proprio come per i casi precedenti, si preferisce utilizzare la variante con tag helper, in quanto consente di mantenere un markup decisamente più comprensibile: nell'esempio ci siamo basati su quanto offerto da Bootstrap, per comporre già un layout ben organizzato.

I metodi in esame estraggono il messaggio di errore dagli attributi di validazione che abbiamo visto in precedenza ma permettono anche di personalizzare il messaggio, passandolo come secondo parametro del metodo.

Come si vede nell'Esempio 7.25 e nella Figura 7.2, il punto migliore dove inserire questi helper di validazione è accanto al campo di input di cui devono mostrare l'errore.

Figura 7.2 – La view con i messaggi di errore.

Esiste tuttavia un altro metodo che mostra tutti i messaggi di errori in una lista.

Il tag helper Validation Summary

Il tag helper Validation Summary ha lo scopo di visualizzare tutti i messaggi di errore in una lista. Questo helper non va visto come alternativa ai metodi ValidationMessage e ValidationMessageFor, bensì come un indispensabile completamento.

Il punto migliore dove mettere la chiamata a Validation Summary è in testa alla pagina, così che i messaggi di errori siano sempre visibili all'inizio. Questo è importante perché la validazione sul client non è affidabile al 100%, in quanto l'utente potrebbe aver disabilitato il JavaScript o perché alcune validazioni possono essere effettuate solo sul server. Nel momento in cui ASP.NET Core MVC riscontra degli errori di validazione, restituisce al client la pagina con i messaggi di errore che, grazie al tag helper Validation Summary, vengono visualizzati in testa alla pagina e non solo accanto ai controlli. In questo modo, l'utente si accorge immediatamente degli errori e l'usabilità del nostro sito migliora sensibilmente.

Utilizzare il tag helper Validation Summary è estremamente semplice in quanto basta solo aggiungere l'attributo `asp-validation-summary` a un tag `div`, come nell'Esempio 7.26

Esempio 7.26

```
<div asp-validation-summary="ModelOnly"></div>
```

Tutti gli attributi e metodi che abbiamo visto finora sfruttano le caratteristiche già presenti nel framework. Nella prossima sezione vedremo come sfruttare il framework di validazione per aggiungere una nostra validazione personalizzata.

Personalizzare la validazione

Gli attributi di validazione presenti in ASP.NET Core MVC coprono le casistiche più comuni, ma non possono ovviamente coprire il 100% dei casi. In questa sezione vedremo come creare un attributo di validazione per il codice fiscale e come poi portare questa validazione anche sul client. Cominciamo dal primo passo, ovvero dalla creazione dell'attributo custom.

Creare un attributo di validazione

Per creare un attributo di validazione, la prima cosa che dobbiamo fare è quella di creare una classe che erediti, direttamente o indirettamente, da `ValidationAttribute`. Una volta creata la classe, dobbiamo eseguire l'override di uno dei due overload del metodo `IsValid`.

Il primo overload prende in input il valore della proprietà (inviato dal client) e restituisce un booleano che specifica se il valore è valido o no.

Il secondo overload accetta in input il valore della proprietà e un oggetto di tipo `ValidationContext` che rappresenta il contesto di validazione e che espone la proprietà `ObjectInstance`, la quale contiene la classe di cui fa parte la proprietà. Questo metodo deve restituire la costante `ValidationResult.Success` se la validazione è andata a buon fine, oppure un oggetto di tipo `ValidationResult`, se la validazione non è andata a buon fine. Il secondo metodo è più completo in quanto ci offre la possibilità di validare un dato anche in base ai valori di altri dati del model, cosa che non potremmo fare con il primo overload.

> *Il primo overload a essere eseguito da ASP.NET Core MVC è quello che accetta il contesto. Se il metodo invoca la versione base, successivamente viene invocato il metodo che accetta solo il valore da validare. Se invece il metodo che accetta il contesto restituisce un risultato, la validazione termina e il metodo che accetta solo il valore non viene invocato.*

Nell'Esempio 7.27 mostriamo il codice necessario alla creazione dell'attributo.

Esempio 7.27

```
public class CodiceFiscaleAttribute : ValidationAttribute
{
   public override bool IsValid(object value)
   {
      var result = false;
      //valida codice fiscale
      return result;
   }
   protected override ValidationResult IsValid(object value,
      ValidationContext validationContext)
   {
      var result = false;
      //valida codice fiscale
      if (result)
         return ValidationResult.Success;
      else
         return new ValidationResult("Codice fiscale errato");
   }
}
```

A questo punto non dobbiamo fare altro che mettere l'attributo sulla proprietà che rappresenta il codice fiscale e, automaticamente, ogni volta che il server ricostruirà il model dai dati provenienti dal client, eseguirà anche la nostra validazione.

Tuttavia, la validazione avviene solo sul server e non sul client. L'usabilità dell'applicazione ne risente in quanto dobbiamo inviare ogni volta i dati al server per verificare l'effettiva validità del codice fiscale. Vediamo ora come portare la validazione sul client.

Aggiungere la validazione lato client

Aggiungere la validazione lato client è probabilmente il processo più complesso di tutto il framework di validazione di ASP.NET Core MVC in quanto, come vedremo più avanti, dobbiamo scrivere sia codice server side sia codice JavaScript per il plugin `jquery.validate`.

Come primo passo, dobbiamo creare una classe che ha il compito di esporre al framework di validazione i parametri da inviare al client per validare il campo di input relativo alla proprietà. Questa classe, chiamata `ModelClientCodiceFiscaleValidationRule`, deve implementare l'interfaccia da `IClientModelValidator` e deve definire un metodo `AddValidation`, che conterrà la logica da utilizzare per generare gli attributi di validazione.

Una volta creata e implementata la classe, dobbiamo esporla al framework di validazione e per fare questo dobbiamo modificare l'attributo `CodiceFiscale`, aggiungendo l'interfaccia `IClientValidatable` e implementando il suo metodo `GetClientValidationRules`. Questo metodo è responsabile di istanziare e valorizzare la classe `ModelClientCodiceFiscaleValidationRule`, includendola quindi nel processo di validazione. Nell'Esempio 7.28 possiamo vedere il codice dell'attributo e della nuova classe.

Esempio 7.28

```
public class CodiceFiscaleAttribute : IClientModelValidator
{
```

```
   ...
   public void AddValidation(ClientModelValidationContext context)
   {
      if (context == null)
      {
         throw new ArgumentNullException(nameof(context));
      }
      MergeAttribute(context.Attributes, ”data-val”, ”true”);
      MergeAttribute(context.Attributes, ”data-val-codicefiscale”,
GetErrorMessage());

      MergeAttribute(context.Attributes, ”data-val-codicefiscale-value”, value);
   }
}
```

A questo punto non dobbiamo far altro che aggiungere alla libreria `jquery.validate` una nuova regola di validazione chiamata `codicefiscale` e creare il metodo che validi il codice fiscale. Per fare questo dobbiamo scrivere nella view in cui eseguiamo la validazione o in un file JavaScript, che poi referenziamo nella stessa view, il codice JavaScript visibile nel prossimo esempio.

Esempio 7.29 – JavaScript

```
jQuery.validator.addMethod(”codicefiscale”,
function (value, element, param) {
   var isValid = false;
   //valida codice fiscale
   return isValid;
});

jQuery.validator.unobtrusive.adapters.add(”codicefiscale”, [],
   function (options) {
      options.rules[”codicefiscale”] = options.params;
      options.messages[”codicefiscale”] = options.message;
});
```

La prima istruzione aggiunge la regola `codicefiscale` alla lista delle regole del plugin `jquery.validate` e specifica il metodo che esegue la validazione. La seconda riga specifica come la regola debba essere interpretata nel caso si utilizzi l’unobtrusive validation (argomento che affronteremo tra poco).

Tutte le forme di validazione che abbiamo visto fino a questo momento riguardano una specifica proprietà del model. Nella prossima sezione vedremo un metodo alternativo per validare un model.

Validazione personalizzata del modello

Quando il model binder comincia a eseguire il codice di validazione, la prima cosa che fa è invocare il metodo `IsValid` degli attributi. Successivamente, il model binder verifica se il modello implementa l’interfaccia `IValidatableObject` e, in caso affermativo, ne invoca il metodo `Validate`. Questo metodo ritorna una lista di oggetti `ValidationResult` ognuno dei quali contiene un errore. Se la lista invece è vuota, allora la validazione è andata a buon fine e il model binder prosegue il suo lavoro. L’Esempio 7.30 mostra un tipo di utilizzo dell’interfaccia `IValidatableObject`.

Esempio 7.30

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
   var minimumYear = DateTime.Today.AddYears(-5).Year;
      if (!IsActive && RegistrationDate.Year < minimumYear)
      {
            yield return new ValidationResult(
                  $”Gli utenti attivi devono essere registrati prima del
{minimumYear}.”,
                  new[] { ”RegistrationDate” });
      }
}
```

Ovviamente, le regole di validazione espresse tramite questa tecnica sono valutate solo sul server e non hanno nessuna interazione con il client, ma sono più potenti, perché consentono di applicare la validazione ad una combinazione di proprietà, piuttosto che alla singola, garantendo di poter concentrare la logica di validazione in un punto solo e applicarla in automatico al modello, a prescindere da dove questo venga poi utilizzato.

Una volta terminato il processo di validazione, il controllo passa alla action ed è in questa fase che dobbiamo controllare il risultato della validazione e decidere cosa fare.

Gestire gli errori nella action

La action che deve processare la richiesta viene sempre eseguita anche se il processo di validazione ha restituito degli errori. Questo significa che la prima cosa che dobbiamo fare è verificare se ci sono stati errori e comportarci di conseguenza. Se ci sono errori, dobbiamo rimandare la view al client, mentre nel caso non ci siano errori dobbiamo salvare i dati e poi reindirizzare a una pagina che dà la conferma dell'avvenuto salvataggio.

Per controllare se ci sono stati errori, dobbiamo interrogare la proprietà `ModelState.IsValid` del controller. Se ci sono errori, la proprietà restituisce `true` e la proprietà `ModelState.Values` contiene la lista degli errori. Ogni elemento della proprietà `ModelState.Values` corrisponde a una proprietà del model ed espone una proprietà `Value` per recuperare il valore della proprietà, e una proprietà `Errors` per recuperare tutti gli errori legati alla proprietà.

Nell'Esempio 7.31, invece, possiamo vedere il codice della action che gestisce la richiesta.

Esempio 7.31

```
[HttpPost]
public IActionResult Create(CustomerModel model)
{
   if (ModelState.IsValid)
   {
      //Salva dati
      return RedirectToAction("Conferma");
   }
   model.Countries.AddRange(GetCountries());
   return View(model);
}
```

Il codice della action è estremamente semplice ma ci sono due cose importanti da notare. La prima è che nel momento in cui c'è un errore e richiamiamo la view, dobbiamo ripopolare la lista degli stati perché questa non può essere ricostruita dal model binder. La seconda è che quando ASP.NET Core MVC esegue il codice della view, questo sfrutta gli errori presenti in `ModelState` per mostrarli subito all'utente.

Nel corso del capitolo abbiamo accennato spesso al model binder. Nella prossima sezione parleremo più approfonditamente di questo componente.

Il model binder

Il model binder è il componente di ASP.NET Core MVC che ha la responsabilità di trasformare i valori di una richiesta in parametri per la action. Il model binder recupera tutti i valori dalla form, dal body di una richiesta AJAX, dal sistema di routing e dalla querystring (esattamente in quest'ordine di priorità) e li mappa sui parametri delle action in base al nome.

Un esempio di come funzioni il model binder è visibile già nel template di default di un progetto ASP.NET Core MVC, in quanto la route di default specifica che è possibile avere un parametro `id` nell'url. Se nella nostra action inseriamo un parametro di nome `id`, il model binder lo valorizza automaticamente con l'id proveniente dal sistema di routing.

Grazie a questo sistema, nelle nostre action non dobbiamo mai preoccuparci di andare a recuperare fisicamente i dati dalla form, querystring o routing, in quanto è il model binder che esegue il lavoro sporco, permettendoci così di lavorare esclusivamente con gli oggetti.

Come abbiamo visto nel corso del capitolo, il model binder è in grado di ricostruire anche oggetti complessi come `CustomerModel`. Questo è possibile perché quando il metodo accetta una classe, il model binder cerca, tra i valori della richiesta, quelli che abbiano come chiave il nome delle proprietà del model. Nel nostro caso, per popolare la proprietà `Name`, il model binder cerca tra i dati della richiesta uno che abbia la chiave con lo stesso nome (e ripete il processo per tutte le proprietà del model).

Nel caso in cui un model abbia proprietà complesse, il model binder è in grado di ricostruire anche quelle, sempre in base a una convenzione. Se, per esempio, il nostro model avesse una proprietà `Address` che è di un tipo composto dalle proprietà `Address`, `City` e `ZipCode`, il model binder cercherebbe nella richiesta un valore con la chiave `Address.Address`, uno con la chiave `Address.City` e uno con la chiave `Address.ZipCode`. La potenza di questa organizzazione risiede nel fatto che il model binder è in grado di ricostruire oggetti annidati all'infinito.

Fortunatamente, gli HTML helper generano codice HTML rispettoso del model binder, in quanto emettono l'attributo HTML `name` così come il model binder se lo aspetta. Questo significa che sfruttando gli HTML helper non dobbiamo preoccuparci della nomenclatura dei campi.

Il model binder può essere esteso e modificato in ogni aspetto. Per esempio, possiamo estendere il model binder per recuperare i dati anche dai cookie o dalla sessione o, ancora, possiamo modificare il modo in cui vengono mappati i dati della richiesta con i parametri della action (aggiungendo regole che vadano oltre il match del nome). Questo argomento non viene trattato in questo capitolo perché verrà approfondito nel Capitolo 15.

Conclusioni

Nel corso del capitolo abbiamo visto come le data annotation ci permettano di inserire dei metadati riguardanti il model e come ASP.NET Core MVC sfrutti questi metadati sia per generare codice sul client sia per eseguire codice sul server. Ne sono un esempio gli attributi di validazione, che vengono sfruttati dagli HTML e dai tag helper per generare codice di validazione sul client, e dal model binder, per eseguire la validazione lato server.

Oltre ad aver visto gli attributi e gli helper, abbiamo anche spiegato come gestire gli errori di validazione sul server e come personalizzare il sistema di validazione sia lato client sia lato server, per avere una perfetta user experience.

Infine, abbiamo visto come ASP.NET Core MVC sia in grado di ricostruire un model partendo dai dati provenienti dal client, così da permetterci di lavorare sempre con oggetti e non con i parametri della richiesta.

Grazie agli helper, alle data annotations e al model binder, possiamo dire che gestire le form in ASP.NET Core MVC è veramente semplice e l'HTML può essere manipolato anche da chi ha maggior dimestichezza con questo che con C#.

Adesso è il momento di passare al prossimo capitolo, nel quale parleremo di come gestire lo stato in maniera più avanzata, per poi proseguire, in quelli successivi, con l'accesso ai dati.

8
La gestione dello stato

Dopo aver analizzato a fondo le form e come gestirne i dati, è tempo di passare a un altro argomento molto importante: come gestire lo scambio d'informazioni tra le varie richieste.

Il colloquio tra client e server è possibile perché esiste uno standard di comunicazione, che è il protocollo HTTP. La sua analisi esula dagli scopi di questo libro ma è, in ogni modo, fondamentale trattarne la caratteristica più importante per quanto riguarda lo sviluppatore web: la sua natura stateless.

L'essere stateless significa che ogni richiesta che il browser invia è processata dal server in maniera completamente scollegata e indipendente dalle precedenti, senza lasciare traccia in quelle successive. Se a prima vista questa può sembrare una limitazione insormontabile, dobbiamo comunque poter gestire lo stato, anche se il protocollo in origine non prevede (e continua a non prevedere) un supporto specifico.

Nel corso degli anni lo sviluppo del Web ha comportato l'evoluzione delle opportunità che questa piattaforma offre. Quello che una volta era semplicemente un insieme di pagine, ora è una vera e propria applicazione, con la necessità di dover collegare logicamente tra loro le pagine visitate da un utente.

ASP.NET mette a disposizione diverse tecniche per gestire lo stato, ognuna delle quali ha una propria area di utilizzo. All'interno di questo capitolo illustreremo queste tecniche e scopriremo quali si adattano meglio alle nostre esigenze.

Come funziona una richiesta HTTP?

Prendiamo come esempio una richiesta HTTP, così da comprendere al meglio come funzioni il meccanismo: il browser comincia invocando la pagina sul server, all'interno del quale viene creato il processo che viene eseguito. Infine, il server invia il risultato al browser, distruggendo il contesto ed eliminando ogni riferimento alla richiesta appena elaborata. Questo meccanismo fa sì che il web sia molto semplice e snello, poiché i server devono sopportare un carico minore, non dovendo mantenere connessioni aperte con i client, una volta che questi hanno ricevuto il risultato.

Per capire bene la differenza tra un protocollo stateless e uno stateful, è bene dare una dimostrazione anche del secondo. L'esempio più classico è quello di una transazione sul database: in questo scenario apriamo una connessione e una transazione, lanciamo le query di aggiornamento e, infine, chiudiamo il colloquio. Durante tutto il tempo, il database e il client sono sempre connessi ed è per questo motivo che si parla di protocollo stateful.

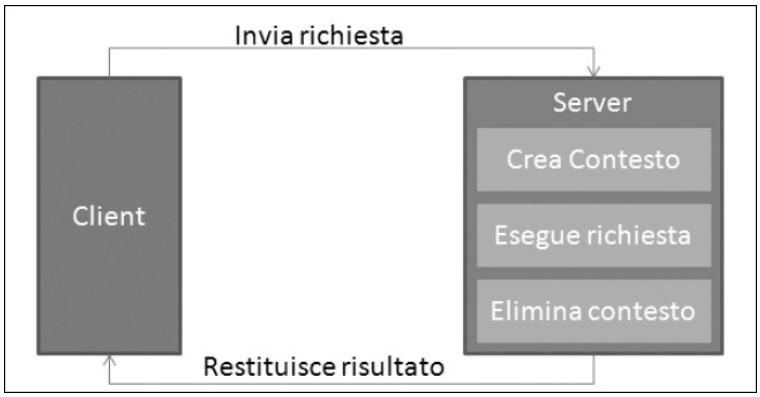


Figura 8.1 – Il flow di una richiesta web.

Da un lato la natura stateless di HTTP garantisce performance ottimali, scalabilità e alta disponibilità ma, dall'altro, rende più difficile lo sviluppo, poiché a ogni richiesta dobbiamo ripartire da zero. Per questo motivo è necessario avere a disposizione una serie di meccanismi per recuperare e salvare le informazioni necessarie al funzionamento dell'applicazione come, per esempio, il nome dell'utente. L'insieme delle problematiche e delle tecniche di persistenza dei dati attraverso le richieste prende il nome di gestione dello stato.

Scenari di gestione dello stato

Esistono diverse tecniche a nostra disposizione per memorizzare informazioni tra le varie richieste. La scelta tra una o l'altra dipende da molti fattori, come la visibilità del dato, la necessità di performance, la sicurezza e la velocità di implementazione.

Per esempio, se dobbiamo salvare temporaneamente l'importo di una fattura, il dato va mantenuto nella pagina che lo gestisce, perché non c'è bisogno che sia visibile altrove. Viceversa, se dobbiamo mantenere il profilo dell'utente finché questo è loggato, non è certo consigliabile utilizzare meccanismi che lo salvano in ogni singola pagina, ma è meglio sfruttarne altri che garantiscono maggior semplicità. E ancora, se abbiamo molti dati da associare a una sessione utente, è preferibile salvarli in un punto dove non occupano risorse

di banda, ma solo spazio sul server. Se le informazioni devono essere persistite, la soluzione ideale è memorizzarle in un punto dove poterle facilmente ritrovare in qualunque momento. Queste casistiche sono molto comuni ed è per questo che, nel corso del capitolo, verranno analizzati in dettaglio i meccanismi di memorizzazione che soddisfano tutte queste esigenze. I meccanismi sono i seguenti:

- i campi hidden, per informazioni che servono su una pagina;
- le sessioni e i cookie per informazioni a livello utente;
- querystring e TempData per i dati da passare da una pagina all'altra;
- la cache per contenere i dati condivisi tra tutti gli utenti.

Vediamo ora in dettaglio le varie tecniche analizzandole una per una.

Lo stato con i campi hidden

Lo standard HTML offre un primo meccanismo per persistere alcune informazioni sulla pagina: i campi hidden. Questi campi sono presenti nel codice HTML inviato al browser, ma sono invisibili all'utente e, al loro interno, possiamo inserire informazioni in formato stringa. Nell'Esempio 8.1 possiamo vedere come inserirne uno all'interno della pagina e come gestirne la valorizzazione.

Esempio 8.1

```
<input type="hidden" asp-for="Id" />
```

Secondo il protocollo HTTP, quando l'utente esegue il post di una form, i campi hidden all'interno della form vengono inviati al server insieme agli altri campi, quindi la proprietà del modello viene valorizzata con il valore del campo hidden.

Grazie a questa tecnica, a ogni post della form possiamo mantenere uno o più valori che possono essere utilizzati. La potenza dei campi hidden risiede nel fatto che possono essere sia letti sia scritti anche sul client tramite JavaScript, come possiamo notare nell'Esempio 8.2.

Esempio 8.2

```
document.querySelector('#@Html.IdFor(m => m.Id)').value = 'new value';
```

I campi hidden hanno dalla loro parte l'estrema semplicità di implementazione, la generazione di pochissimo codice HTML e la possibilità di accesso ai valori direttamente nel browser, utilizzando JavaScript.

Il rovescio della medaglia è che essendo i campi hidden visibili nel codice HTML inviato dal server, essendo modificabili lato client e, soprattutto, inviati in chiaro, non possono essere utilizzati per trasportare dati sensibili come numeri di carta di credito, password o altro ancora, e nemmeno si può fare affidamento sul fatto che questi dati non siano modificabili tra una richiesta e l'altra. Qualunque utente con un tool di tracing HTTP (oramai integrato in tutti i browser) può modificare il valore di un campo hidden prima che questo venga inviato al server. Per questo motivo, è bene sempre validare lato server i dati trasportati dai campi hidden, esattamente come si fa per tutti gli altri campi di input.

I campi hidden sono utili per trasportare valori all'interno di una form, ma perdono significato quando dobbiamo persistere dati di un utente durante la sua navigazione nell'applicazione. In questi casi possono tornare utili i cookie.

Lo stato attraverso i cookie

Per risolvere il problema relativo al salvataggio dei dati necessari durante tutta la sessione utente, la prima soluzione in ordine di apparizione nel mondo web è stata quella basata sui cookie.

Un cookie è un file di testo che contiene una coppia chiave-valore che può essere utilizzata dal server per diversi scopi. L'utilizzo più comune dei cookie (come scopriremo nel Capitolo 17) è quello di autenticare una richiesta, di mantenere alcune opzioni selezionate dall'utente, di memorizzare temporaneamente il percorso di navigazione e così via. Essendo un semplice file di testo, in esso possiamo immagazzinare solamente dati primitivi, come stringhe, interi e booleani. Per renderne più evoluto l'utilizzo, possiamo impostare alcune proprietà, come il dominio e il percorso, allo scopo di avere un accesso più ristretto e sicuro, la data di scadenza, oltre la quale il cookie non è più valido o, ancora, l'accessibilità e il tipo di trasferimento tra browser e server.

Prima di parlarne in maniera dettagliata, è bene che spendiamo qualche parola su come questi file sono creati e su come viaggiano tra client e server. Il cookie viene generato tramite codice sul server, per essere spedito al browser insieme alla pagina in cui viene creato. Il browser salva in locale il file e, a ogni nuova richiesta, lo rispedisce al server, che lo recupera e lo utilizza in base al codice. Se il cookie viene aggiornato, la nuova versione viene rispedita al client; in caso negativo non c'è alcuna trasmissione, risparmiando così banda che, altrimenti, sarebbe sprecata. Grazie a questo meccanismo, il cookie è sempre disponibile sul server e, di conseguenza, i suoi dati sono pronti a ogni richiesta, senza bisogno di codice aggiuntivo.

> *Un cookie può anche essere generato sul client tramite JavaScript. Una volta generato, questo cookie segue la stessa strada di tutti gli altri cookie. Questa tecnica è poco usata in quanto molte applicazioni bloccano questo tipo di cookie.*

Per manipolare i cookie dobbiamo innanzitutto accedere all'oggetto che rappresenta la risposta della richiesta che stiamo processando. Se siamo all'interno di un controller basta accedere alla proprietà `Response`, mentre se siamo in una classe esterna possiamo iniettare l'interfaccia `IHttpContextAccessor` e accedere alla sua proprietà `Response`. Una volta ottenuto l'oggetto, dobbiamo accedere alla sua proprietà `Cookies` (di tipo `IResponseCookies`) e utilizzare i suoi metodi:

- `Append`: crea un cookie sul client o lo sovrascrive se esiste già.
- `Delete`: elimina un cookie sul client;

Il metodo `Append` prende in input due stringhe, la prima rappresenta la chiave del cookie e la seconda il valore, mentre il metodo `Delete` prende in input una sola stringa che rappresenta la chiave del cookie da eliminare.

Esempio 8.3

```
Response.Cookies.Append(key, value);
Response.Cookies.Delete(key);
```

`Append` ha un overload che accetta un terzo parametro di tipo `CookieOptions` che permette di specificare i parametri aggiuntivi per il cookie che vogliamo scrivere. La Tabella 8.1 mostra le proprietà di questa classe e il loro scopo.

Tabella 8.1 – Le proprietà della classe CookieOptions.

Attributo	Descrizione
`Domain`	Specifica il dominio a cui il cookie è associato. Per default il cookie è associato al dominio che lo ha scritto, ma può essere reso visibile anche ad altri sottodomini, impostando il valore di questa proprietà al nome del sottodominio.
`Expires`	Specifica la data dopo la quale il cookie scade. Una volta raggiunta questa data, il browser elimina il cookie. Se non si imposta una data di scadenza, il cookie viene automaticamente cancellato alla chiusura del browser.
`HttpOnly`	Specifica se un cookie può essere letto o modificato dal client.
`IsEssential`	Specifica se la presenza del cookie è obbligatoria. Questa proprietà viene usata dai controlli GDPR, di cui parleremo nel Capitolo 18.
`MaxAge`	Specifica la durata del cookie, impostando l'header HTTP max-age.
`Path`	Specifica il percorso all'interno del quale il cookie è disponibile. Per default il cookie è disponibile in tutta l'applicazione, ma attraverso questa proprietà possiamo specificare che sia disponibile solo in una data sottocartella.
`SameSite`	Specifica la policy del cookie per prevenire attacchi di tipo CSRF. Maggiori informazioni possono essere reperite a questo indirizzo: http://aspit.co/ai9.
`Secure`	Specifica che il cookie deve viaggiare solo se la connessione è HTTPS.

Come abbiamo detto in precedenza, il client invia i cookie al server a ogni richiesta. Per leggerli dobbiamo accedere alla richiesta tramite la proprietà `Request` del controller o tramite l'iniezione dell'interfaccia `IHttpContextAccessor` al di fuori del controller. Una volta ottenuta la richiesta, dobbiamo accedere alla proprietà `Cookies` (di tipo `IRequestCookieCollection`) e sfruttare i suoi membri per recuperare i cookie di cui elenchiamo qui quelli principali:

- `Keys`: proprietà che torna la lista delle chiavi dei cookie.
- Indexer: torna il valore del cookie data la sua chiave. Solleva un'eccezione se il cookie non esiste.
- `TryGetValue`: torna il valore del cookie data la sua chiave, senza sollevare un'eccezione se la chiave non esiste.

Oltre a poter usare usare i suoi membri, poiché il tipo `IRequestCookieCollection` eredita da `IEnumerable<KeyValuePair>` possiamo anche enumerare i cookie con un normale ciclo. Nel prossimo esempio vediamo i diversi modi di leggere i cookie.

Esempio 8.4

```
public IActionResult ReadCookies()
{
   var model = new Dictionary<string, string>();
   foreach(var item in Request.Cookies)
   {
      model.Add(item.Key, item.Value);
   }
   var value = Request.Cookies["CookieKey"];
   Request.Cookies.TryGetValue("CookieKey", out var tryvalue);
   return View(model);
```

```
}
```

Il ciclo foreach enumera i cookie inviati dal client, l'indexer cerca il valore di uno specifico cookie così come il metodo TryGetValue.

Il vantaggio dell'utilizzo dei cookie è l'estrema semplicità con cui le informazioni possono essere recuperate, modificate e immagazzinate, in quanto la persistenza e la disponibilità sono gestite automaticamente da browser e server, senza il nostro intervento. In aggiunta, il server non deve mantenere alcuna risorsa per salvare questi dati, risparmiando quindi memoria.

Tuttavia anche i cookie presentano i loro svantaggi. Un singolo cookie ha una dimensione massima di 4096 byte, il che significa che per memorizzare grandi valori dobbiamo spezzarli in più cookie. Inoltre, viaggiando sempre avanti e indietro tra client e server, i cookie possono consumare molta banda, rallentando la nostra applicazione specie se usufruita da dispositivi mobili in scarsa connettività. Per concludere, i cookie possono conservare solo dati semplici, quindi, se vogliamo memorizzare dati complessi, dobbiamo preoccuparci di serializzarli e deserializzarli rispettivamente prima della scrittura e della lettura. Per questi motivi i cookie non sono sempre la scelta migliore e il loro utilizzo va valutato in base alle esigenze.

Nel prossimo paragrafo, prenderemo in esame una tecnica che sfrutta le sole risorse del server per mantenere lo stato: la sessione.

Gestione dello stato nella sessione

Quando ci connettiamo a un sito web, il server crea una sessione associata al browser. A ogni sessione corrisponde un'area di memorizzazione, all'interno della quale possiamo salvare dati relativi a un utente, per tutta la durata del suo percorso di navigazione: questi dati prendono il nome di sessione. A differenza dei cookie, queste informazioni risiedono sul server e non sul client, riducendo così le necessità di banda.

Il luogo dove vengono memorizzate le informazioni è stabilito da un provider. Per default ASP.NET Core utilizza la RAM del server che esegue la richiesta, ma possiamo modificare questo comportamento per salvare i dati di sessione sul database, su una cache o altro ancora.

Per capire come faccia il server ad associare quest'area di memoria a uno specifico browser, dobbiamo andare ad analizzare l'inizio della comunicazione. Quando un browser richiede una pagina per la prima volta, il server genera dinamicamente un identificativo univoco, per contrassegnare la sessione appena aperta. A questo punto il server invia al client, insieme alla pagina richiesta, anche un cookie, detto cookie di sessione, contenente al suo interno l'identificativo.

Come abbiamo visto nel paragrafo precedente, a ogni richiesta il browser trasmette il cookie al server, che si serve dell'identificativo in esso contenuto per recuperare dalla memoria i dati associati. Il processo di salvataggio e recupero è completamente trasparente per noi, in quanto se ne occupa il middleware SessionMiddleware che iniettiamo nella pipeline di esecuzione tramite il metodo UseSession nel metodo Configure di Startup. Il metodo UseSession può essere invocato senza parametri oppure passando un oggetto di tipo SessionOptions che permette di specificare i parametri del cookie e alcuni parametri di comunicazione con lo store che memorizza i dati in sessione. I parametri relativi al cookie sono quelli visti nella sezione precedente mentre quelli relativi alla connessione con lo store sono invece molto interessanti. Il primo è IOTimeout che specifica il timeout di lettura e scrittura della sessione nello store che mantiene i dati. Il secondo è IdleTimeout che specifica dopo quanto tempo dall'ultimo accesso alla sessione questa può essere eliminata dallo store (il valore di default è 20 minuti).

Dopo aver configurato la pipeline, aggiungiamo la sessione ai servizi, usando il metodo AddSession nel metodo ConfigureServices di Startup. AddSession aggiunge il servizio DistributedSessionStore che a sua volta dipende da IDistributedCache (che rappresenta lo store della sessione), quindi dobbiamo aggiungere anche questo servizio alla dependency injection, invocando il metodo AddDistributedMemoryCache. Il nome del metodo è abbastanza esplicativo: la sessione viene memorizzata in un'area dedicata della cache, a prescindere da quale sia lo store della cache.

Anche il metodo AddSession permette di configurare le opzioni della sessione così come UseSession. AddSession accetta un parametro opzionale, che è un metodo che accetta in input l'oggetto SessionOptions e all'interno del quale possiamo impostarne le proprietà.

Esempio 8.5

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddDistributedMemoryCache();
   //senza parametri
   services.AddSession();
   //con configurazione delle opzioni
   services.AddSession(a => a.IdleTimeout = TimeSpan.FromMinutes(5));
}

public void Configure(IApplicationBuilder app,
   IHostingEnvironment env)
{
   //senza parametri
```

```
    app.UseSession();
    //con configurazione delle opzioni
    app.UseSession(new SessionOptions {
       IdleTimeout = TimeSpan.FromMinutes(5))
    }
 }
```

Una volta configurata la sessione, non ci rimane che leggere e scrivere dati al suo interno. Per fare questo, dobbiamo prima recuperare l'oggetto che ci permette di accedere alla sessione relativa alla richiesta web e poi sfruttare i suoi metodi. Per ottenere l'oggetto dobbiamo prima recuperare il contesto della richiesta (tramite la proprietà `HttpContext` del controller o tramite l'injection dell'interfaccia `IHttpContextAccessor`) e poi accedere alla proprietà `Session` di tipo `ISession`. I membri principali esposti da quest'interfaccia sono i metodi `TryGetValue` e `Set` che rispettivamente recuperano e leggono un valore dalla sessione, i metodi `Remove` e `Clear` che rispettivamente rimuovono uno o tutti gli elementi dalla sessione, la proprietà `IsAvailable` che specifica se la sessione è stata caricata dallo store.

Quando utilizziamo i metodi `TryGetValue` e `Set`, il valore che leggiamo e scriviamo non è il valore nell'oggetto originale, ma il valore serializzato in un array di byte. Serializzare e deserializzare ogni volta il valore è un'operazione ripetitiva, ma possiamo limitare le ripetizioni usando gli extension method `GetString`, `GetInt32`, `SetString` e `SetInt32` che lavorano nativamente con stringhe e interi.

Esempio 8.6

```
public IActionResult Session()
{
   HttpContext.Session.SetString("key", "value");
   var value = HttpContext.Session.GetString("key");
   HttpContext.Session.Remove("key");
}
```

Nei progetti reali, è comune salvare in sessione non solo interi e stringhe ma anche oggetti complessi. Per salvare questo tipo di oggetti, possiamo prima serializzarli in una stringa JSON e poi usare il metodo `SetString`. Nel momento di leggere il valore facciamo il discorso inverso: usiamo il metodo `GetString` per recuperare la stringa JSON e poi la deserializziamo in un'oggetto. Nel prossimo esempio creeremo due extension method che mettono in pratica quanto detto.

Esempio 8.7

```
public static class SessionExtensions
{
   public static void SetValue(this ISession session, string key,object value)
   {
      session.SetString(key, JsonConvert.SerializeObject(value));
   }
   public static T GetValue<T>(this ISession session, string key)
   {
      var value = session.GetString(key);
      return value == null ?
         default(T) :
         JsonConvert.DeserializeObject<T>(value);
   }
}
```

Questi metodi vanno a rimpiazzare gli altri metodi di lettura e scrittura nel codice, in quanto permettono di gestire qualunque tipo di dato.

> *Come abbiamo detto in precedenza, la sessione si basa sullo store della cache che implementa l'interfaccia IDistributedCache. Il metodo AddDistributedMemoryCache aggiunge uno store che salva la sessione in memoria, ma questo non è l'unico metodo disponibile. Esistono i metodi AddDistributedSqlServerCache e AddDistributedRedisCache che configurano rispettivamente Sql Server e Redis come store. Questi metodi verranno analizzati più avanti nel capitolo, quando parleremo della cache. Per ora basti sapere che qualunque provider utilizziamo per la cache, questo verrà utilizzato anche per la sessione.*

Le sessioni sono sicuramente più semplici da utilizzare rispetto ai cookie. Inoltre offrono un risparmio di banda, in quanto risiedono sul server e quindi non viaggiano avanti e indietro. Comunque, se da un lato il carico di rete diminuisce, dall'altro aumenta l'utilizzo di risorse sul server oltre allo svantaggio che se l'utente chiude il browser, gli oggetti in sessione vengono persi perché il cookie che contiene l'id della sessione non è persistente. Emettendo un cookie, invece, si ha l'opportunità di decidere se deve essere persistente o no, e quindi questo può sopravvivere alla chiusura del browser.

Dobbiamo inoltre tenere a mente che nei casi in cui abbiamo più macchine in load balancing, la sessione deve sfruttare uno store esterno e non la memoria della macchina che sta eseguendo la richiesta. Se rimanesse nella memoria di una singola macchina, avremmo

dei comportamenti non previsti, in quanto successive richieste potrebbero essere evase da un'altra macchina in load balancing e questa non avrebbe le variabili di sessione in memoria.

Per chiudere, dobbiamo considerare che nella sessione possiamo scrivere solo dati serializzabili il che, in alcuni scenari, può essere una limitazione (vedi una connessione TCP, una connessione al database o altro ancora).

Prima di scegliere la sessione è bene prendere in considerazione tutte queste sfaccettature. Solo dopo aver valutato attentamente tutte le limitazioni, allora possiamo affidarci a questa tecnica. Va tuttavia detto che l'utilizzo della sessione è sempre un'ultima spiaggia da scegliere solo quando non ci sono altre strade.

Finora abbiamo visto come gestire lo stato per persistere i dati mentre ci si trova su una pagina o per tutta la navigazione. Ora vediamo come gestire dati temporanei che vivono solo durante la richiesta.

Passare valori tramite querystring

La querystring è la parte dell'url che segue il carattere `?`. In questa parte sono inseriti tutti i parametri che vogliamo passare in input a una richiesta web nel formato `{chiave}={valore}` dove ogni coppia è separata dal carattere `&`. Un tipico esempio di url con querystring è: http://www.sito.it/home/qs?key=1&v=A. In questo caso, la action che viene invocata da questo url riceve in input i parametri `key` e `v` che hanno rispettivamente i valori `1` e `A`.

I valori in querystring vivono per il ciclo di vita della richiesta. Se vogliamo persistere il valore di queste variabili così che siano disponibili su più richieste, dobbiamo o aggiungerli alla querystring di ogni url (scomodo quando possono esserci molte richieste che necessitano del parametro) oppure salvarli in uno store che può essere un cookie, in sessione, sul database o ancora altrove.

Per generare un url con querystring, tutto quello che dobbiamo fare è utilizzare il metodo `Url.Action`, mentre se vogliamo generare un tag `a` possiamo usare il metodo `Html.ActionLink` o il tag helper `a`. Questi oggetti sono già stati analizzati nei Capitoli 6 e 7, quindi non ci torneremo nuovamente. In questa sezione, invece, vedremo come sfruttare questi parametri nel nostro codice server.

Come abbiamo detto, la action riceve in input i parametri in querystring. Per recuperarli, abbiamo due possibilità: la prima è passare alla action tanti parametri quante sono le chiavi in querystring, facendo attenzione a impostare il nome del parametro con il nome della chiave; la seconda, invece, prevede l'accesso alla proprietà `Request` (di tipo `HttpRequest`) del controller o, se ci troviamo fuori dal controller, alla proprietà `Request` dell'oggetto `HttpContext` recuperabile tramite iniezione dell'interfaccia `IHttpContextAccessor`). `HttpRequest` espone la proprietà `QueryString`, che dà accesso all'intera stringa di querystring, e la proprietà `Query`, che permette di accedere ai singoli valori della querystring tramite chiave. L'Esempio 8.8 mostra entrambe le tecniche.

Esempio 8.8

```
public IActionResult QS(int key, string v)
{
   var model = new QSModel();
    model.KeyFromParam = key;
    model.VFromParam = v;
    model.KeyFromQS = Request.Query["key"];
    model.VFromQS = Request.Query["v"];
}
```

Quando siamo all'interno di una action, sicuramente l'utilizzo dei parametri è più semplice; se invece ci troviamo in una classe esterna alla action, come un filter, un middleware o altro ancora, ricorrere alla `Request` è la sola scelta possibile.

In querystring possono essere passati solo valori semplici come numeri, stringhe o booleani. Nel caso vogliamo passare valori più complessi, come un oggetto, dobbiamo prima serializzarlo in una stringa e poi passarlo in querystring come tale. Sarà poi compito di chi riceve il valore deserializzarlo nel tipo corretto.

> *Non esiste un limite per la lunghezza di un url, quindi in teoria possiamo mettere in querystring quanti dati vogliamo. Tuttavia non essendoci specifiche, ogni browser può imporre un proprio limite e anche i web server possono avere un loro limite di lunghezza. Per questi motivi è meglio evitare di mettere troppi dati in querystring (1-2 KB, per dare un'idea di massima).*

Vediamo ora un altro meccanismo di gestione dello stato che prevede la memorizzazione di un dato dalla scrittura fino alla prima lettura.

Gestione dei dati temporanei con TempData

Nella maggior parte dei casi, quando un utente si trova davanti a una form e preme il tasto di salvataggio dati, il server memorizza i dati e poi reindirizza l'utente a una pagina che dà la conferma dell'avvenuto salvataggio. Oltre a dare conferma, potremmo anche mostrare all'utente alcuni dati riepilogativi (come un numero di prenotazione, una email un numero di telefono o qualunque altra informazione). Queste informazioni possono essere passate tramite querystring oppure tramite TempData. Il TempData è uno store che contiene un dictionary chiave-valore dove si possono inserire chiavi infinite, ma queste vengono eliminate la prima volta che vengono lette.

Nei casi in cui dobbiamo passare dati temporanei da una pagina all'altra, il TempData può essere una buona scelta rispetto alla querystring quando i dati da passare sono molti (il TempData è memorizzato in un cookie, quindi ha meno restrizioni della querystring) o quando non vogliamo far passare in chiaro i dati (il TempData è basato su provider, quindi possiamo memorizzarne i dati sul server, se necessario).

Il provider di default per il TempData è basato su cookie ed è aggiunto ai servizi quando aggiungiamo MVC, quindi non dobbiamo configurare null'altro. Se invece vogliamo usare il provider che salva i dati in sessione, dobbiamo configurare sia il provider (tramite il metodo `AddSessionStateTempDataProvider`) sia la sessione (come visto in precedenza). L'Esempio 8.9 mostra il codice del metodo `ConfigureServices` necessario a configurare il provider per la sessione.

Esempio 8.9

```
public void ConfigureServices(IServiceCollection services)
{
   services
      .AddMvc()
      .AddSessionStateTempDataProvider();
   services.AddSession();
}
```

Per una corretta esecuzione, prima del metodo `AddSession` è fondamentale invocare il metodo `AddSessionStateTempDataProvider`.

Passato lo step di configurazione, possiamo scrivere e leggere i dati dal TempData sfruttando l'omonima proprietà, di tipo `ITempDictionary`, del controller. Per aggiungere un valore, usiamo il metodo `Add` passando la chiave e il valore, mentre per recuperare il valore usiamo l'indexer. Se vogliamo leggere un valore senza che questo venga eliminato, dobbiamo usare il metodo `Peek`.

Esempio 8.10

```
public IActionResult TempDataAdd()
{
   TempData.Add("Key", "value");
   return Content(String.Empty);
}

public IActionResult TempDataPeek()
{
   var value = TempData.Peek("Key");
   return Content((string)value);
}

public IActionResult TempDataGet()
{
   var value = TempData["Key"];
   return Content((string)value);
}
```

La prima action che invochiamo è `TempDataAdd`, tramite la quale aggiungiamo un valore in TempData. Successivamente, possiamo invocare la action `TempDataPeek` infinite volte: questa restituisce sempre il valore in TempData, in quanto usando il metodo `Peek` il valore non viene rimosso. Infine, se invochiamo il metodo `TempDataGet`, la prima volta questo restituisce il valore e cancella il valore, quindi se lo invochiamo una seconda volta torna null.

L'utilizzo di TempData non è sempre semplice da identificare e molto spesso capita di non usarlo nei progetti, a favore della querystring o di altri meccanismi. Il suo uso, come sempre, dipende dalle necessità dell'applicazione.

Mantenere dati di una richiesta: HttpContext.Items

Durante il ciclo di vita di una richiesta web, possiamo avere la necessità di memorizzare dati relativi solo a quella richiesta e di doverli distruggere a fine richiesta. Supponiamo di dover misurare il tempo di esecuzione di una richiesta web. Il modo migliore è quello di far partire un timer all'inizio della richiesta e fermarlo alla fine, per misurare il tempo. Se poi vogliamo misurare anche quanto tempo impiegano alcuni punti intermedi della richiesta, dobbiamo recuperare il timer e prendere nuovamente i tempi.

Il posto migliore dove mettere il timer è nella proprietà `HttpContext.Items`. Questa proprietà espone un dictionary, di tipo `IDictionary`, all'interno del quale possiamo mettere oggetti che vivono per tutto il ciclo della richiesta. L'Esempio 8.11 mostra come utilizzare `HttpContext.Items` per memorizzare il timer e come recuperarlo in una action.

Esempio 8.11

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   // Aggiunge un middleware che crea il timer
```

```
   app.Use(async (context, next) =>
   {
      var sw = new Stopwatch();
      context.Items[”sw”] = sw;
      sw.Start();
      await next.Invoke();
      sw.Stop();
      Debug.WriteLine(sw.ElapsedMilliseconds);
   });
}

// Crea una action che legge il timer
public IActionResult HttpItems()
{
   var sw = (Stopwatch)HttpContext.Items[”sw”];
   return Content(sw.ElapsedMilliseconds.ToString());
}
```

Come abbiamo visto in altre occasioni, oltre che all’interno di un controller, possiamo accedere ad `HttpContext` anche iniettando l’interfaccia `IHttpContextAccessor` nel costruttore di una classe esterna.

Adesso cambiamo argomento e vediamo l’ultima tecnica in grado di mantenere lo stato di un’applicazione: la cache.

Utilizzare la cache

La cache è un’area di memoria che mantiene dati disponibili in tutta l’applicazione. All’interno della cache possiamo mettere qualunque tipo di dato, dal risultato di una query al codice HTML di una pagina, dal contenuto di un file ai parametri di sistema o altro ancora. Lo scopo principale della cache è quello di essere performante e di garantire un accesso veloce ai dati. Per questo l’utilizzo della cache può essere una delle chiavi vincenti nel mantenere un ottimo livello di performance delle nostre applicazioni.

In ASP.NET la cache è suddivisa in tre funzionalità principali: data caching, HTML caching e response caching. La prima funzionalità prevede il salvataggio di dati all’interno della cache, la seconda prevede il salvataggio di intere porzioni di codice HTML in cache e la terza prevede la creazione di intestazioni HTTP che istruiscono client, web server e proxy intermedi su come mantenere nella propria cache l’intero risultato di una richiesta. Le prime due funzionalità coinvolgono interamente ASP.NET e la cache, mentre l’ultima non ha alcun impatto sulla cache, in quanto è solo un modo per indicare alle varie parti interessate da una richiesta come memorizzare una risposta HTML.

A prescindere dalla tipologia di dato che mettiamo in cache, ASP.NET offre due tipi di cache: una cache locale e una cache distribuita. La prima prevede uno store che memorizza tutti i dati in locale sulla macchina, mentre la seconda prevede la possibilità che lo store sia in un servizio esterno alla macchina e quindi condiviso tra tutte le macchine che fanno parte della web farm. Per ogni tipologia di cache ASP.NET offre classi e metodi di configurazione già pronti per l’utilizzo.

Per la cache locale ASP.NET mette a disposizione l’interfaccia `IMemoryCache` e la sua implementazione `MemoryCache`.

Per la cache distribuita ASP.NET mette a disposizione l’interfaccia `IDistributedMemoryCache` e tre diverse implementazioni:

- `SqlServerCache`: memorizza la cache su SqlServer.
- `RedisCache`: memorizza la cache su un server Redis.
- `MemoryDistributedCache`: memorizza la cache in locale sulla macchina.

Il fatto che la distributed cache abbia un’implementazione che usa uno store locale come la local cache fa spesso propendere per l’utilizzo della distributed cache anche in scenari dove non è presente una web farm. Questo perché se un giorno l’applicazione dovesse crescere e necessitare di più macchine, sarebbe estremamente semplice cambiare il provider da quello alla memoria in locale a uno fisicamente distribuito come quello per SqlServer o Redis.

Tuttavia, vale la pena parlare sia del provider di local cache sia di quello di distributed cache, a partire dal primo.

Local cache

La local cache espone diverse funzionalità che ne rendono l’utilizzo estremamente semplice e anche potente. Infatti, ogni elemento aggiunto in cache può avere una scadenza assoluta (absolute expiration) oppure una scadenza relativa alla data dell’ultimo accesso (sliding expiration), può essere marcato per non essere mai rimosso, può scatenare un evento alla rimozione e può dipendere da un altro elemento in cache. Da questo si deduce che la cache non è un semplice contenitore di coppie chiave-valore, ma un oggetto con una logica molto complessa.

Come abbiamo visto in precedenza, la local cache è un servizio e per poterlo utilizzare dobbiamo configurarlo invocando il metodo AddMemoryCache nel metodo ConfigureServices della classe Startup.

Il metodo AddMemoryCache accetta opzionalmente un metodo all'interno del quale specifichiamo le opzioni relative all'uso della cache tramite un oggetto di tipo MemoryCacheOptions. Questo tipo espone due proprietà importanti: SizeLimit, che indica la quantità massima di dati in cache (in KB) e ExpirationScanFrequency, che indica ogni quanto tempo effettuare lo scan in memoria.

Esempio 8.12

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMemoryCache(o => {
      // Imposta opzioni cache
   });
}
```

A questo punto ci basta iniettare l'interfaccia IMemoryCache nel controller, nella action o in un'altra classe istanziata dal motore di dependency injection, e siamo liberi di utilizzare i suoi metodi che sono tre: CreateEntry per aggiungere un valore in cache, Remove per eliminarlo e TryGetValue per recuperarlo.

I metodi CreateEntry e TryGetValue sono di basso livello e, per questo motivo, l'interfaccia è arricchita di extension methods che ne semplificano l'uso. Gli extension method sono Get, Set e GetOrCreateAsync e analizzeremo solo questi oltre a Remove.

Com'è facile immaginare, il metodo Set imposta un elemento all'interno della cache: se l'elemento non esiste lo crea, se già esiste lo sovrascrive. Questo metodo accetta la chiave e il valore da inserire, ma ha anche altri overload che permettono di specificare la scadenza, le dipendenze e altro ancora. Il primo overload accetta un parametro di tipo DateTimeOffset, che rappresenta l'absolute expiration, mentre il secondo accetta un parametro di tipo TimeSpan, che rappresenta la sliding expiration. Il terzo overload accetta un oggetto di tipo MemoryCacheEntryOptions, che specifica tutte le possibili opzioni per l'elemento, comprese quelle specificate negli altri overload.

Esempio 8.13

```
public IActionResult Cache([FromServices] IMemoryCache cache)
{
   // elemento senza opzioni
   cache.Set("key", "value");
   // elemento che scade dopo un'ora
   cache.Set("key", "value", DateTime.Now.AddHours(1));
   // elemento che scade dopo 10 minuti dall'ultimo accesso
   cache.Set("key", "value", TimeSpan.FromMinutes(10));
   // elemento che scade dopo 10 minuti dall'ultimo accesso
   cache.Set("key", "value", new MemoryCacheEntryOptions {
      SlidingExpiration = TimeSpan.FromMinutes(10) });
}
```

Il metodo Get ritorna il valore data la sua chiave. Questo metodo restituisce il valore come object ma ha anche un overload che accetta un parametro generico. Se usiamo questo overload, il tipo del valore restituito è quello del parametro generico.

Esempio 8.14

```
object value = cache.Get("key");
string stringValue = cache.Get<string>("key");
```

Il metodo GetOrCreate accetta in input la chiave del valore da recuperare e un metodo da invocare qualora il valore non esista e che ritorna il valore dell'elemento da aggiungere in cache con le relative opzioni. Una volta inserito, il valore viene quindi restituito come risultato della chiamata. Nel caso in cui per recuperare il valore da mettere in cache dobbiamo eseguire una chiamata asincrona (query sul database, chiamata HTTP o altro ancora, possiamo usare la versione asincrona GetOrCreateAsync. Nell'Esempio 8.15 aggiungiamo il metodo per recuperare il valore dell'elemento con chiave Key: se l'elemento non esiste, viene invocato il secondo parametro, che torna il valore value che verrà inserito in cache con una durata di 10 minuti.

Esempio 8.15

```
var value = cache.GetOrCreate("key", e =>
{
   e.AbsoluteExpiration = DateTime.Now.AddMinutes(10);
   return "value";
});

var valueAsync = await cache.GetOrCreateAsync("key", e =>
```

```
{
   e.AbsoluteExpiration = DateTime.Now.AddMinutes(10);
   return SomeAsyncMethod();
});
```

Il metodo `Remove` elimina un elemento dalla cache data la sua chiave. Se la chiave non esiste, il metodo non va in errore ma continua la sua esecuzione.

Esempio 8.16

```
cache.Remove("key");
```

Ora che abbiamo visto come manipolare gli oggetti, vediamo come sfruttare le potenzialità della cache per gestire le funzionalità che abbiamo menzionato all'inizio di questa sezione.

Gestione avanzata della local cache

Abbiamo già visto che per impostare funzionalità avanzate sugli elementi della cache dobbiamo sfruttare i membri della classe `MemoryCacheEntryOptions`. Abbiamo già visto che possiamo impostare l'absolute expiration sfruttando la proprietà `AbsoluteExpiration` (o il metodo `SetAbsoluteExpiration`) e che possiamo anche impostare la sliding expiration tramite la proprietà `SlidingExpiration` (o il metodo `SetSlidingExpiration`).

Quando il server comincia a essere a corto di memoria, parte una procedura in background che rimuove automaticamente gli elementi dalla cache. Se abbiamo elementi che preferiamo rimangano in cache (perché acceduti molto spesso) rispetto ad altri, possiamo dare a questi elementi una priorità maggiore, così che il processo elimini altri elementi con priorità minore. La priorità è assegnata impostando la proprietà `Priority` (un enum di tipo `CacheItemPriority`) o invocando il metodo `SetPriority`.

Se vogliamo essere notificati quando un elemento viene rimosso (per scopi di logging o perché vogliamo ricaricarlo immediatamente), possiamo impostare un callback tramite la proprietà `PostEvictionCallbacks` (che rappresenta una lista di callback) o invocando il metodo `RegisterPostEvictionCallback` (che aggiunge il callback alla lista).

Infine, a ogni elemento possiamo aggiungere una o più dipendenze. Ogni volta che una di queste diendenze solleva una notifica, tutti gli elementi legati alla dipendenza vengono rimossi. Una dipendenza è una classe che implementa l'interfaccia `IChangeToken` e .NET Core ne offre due già pronte: `PollingFileChangeToken`, che monitora i cambiamenti a un file e invia una notifica per ogni modifica, e `CancellationChangeToken`, che rappresenta una chiave in cache contenente un `CancellationTokenSource` e che invia una notifica quando viene chiamato il metodo `Cancel` del token. Nel prossimo esempio vediamo come impostare una chiave che sfrutta tutti i meccanismi visti in questa sezione.

Esempio 8.17

```
public IActionResult Cache()
{
   var cts = new CancellationTokenSource();
   cache.Set("keyDep", cts);
   var value = cache.GetOrCreate("key", e =>
   {
      e.SetAbsoluteExpiration(DateTime.Now.AddMinutes(10));
      e.SetSlidingExpiration(TimeSpan.FromMinutes(1));
      e.ExpirationTokens.Add(new CancellationChangeToken(cts.Token));
      e.ExpirationTokens.Add(new PollingFileChangeToken(new
         FileInfo(@"c:\file.txt")));
      e.RegisterPostEvictionCallback(OnEviction);
      return "value";
   });
}

public IActionResult CacheExpire()
{
   cache.Get<CancellationTokenSource>("keyDep").Cancel();
   return Content("cancelled");
}

private void OnEviction(object key, object value,
   EvictionReason reason, object state)
{
   Debug.WriteLine($"{key} removed. Reason:{reason}");
}
```

In questo esempio impostiamo una scadenza assoluta e una basata sulla data di accesso, aggiungiamo una scadenza basata sulle modifiche a un file, una scadenza basata su una chiave della cache che contiene un cancellation token e configuriamo anche un callback da invocare

quando l'elemento viene eliminato dalla cache. Infine, nella action `CacheExpire`, vediamo anche come usare il cancellation token usato come dipendenza per far scadere tutti gli elementi collegati.

Distributed cache

La distributed cache offre meno funzionalità rispetto alla local cache. Tuttavia rimane una scelta obbligata in scenari di web farm, quindi è bene conoscerla approfonditamente. Abbiamo già visto nella sezione relativa alla sessione che, per configurare il provider che utilizza come store la memoria della macchina, dobbiamo usare il metodo `AddDistributedMemoryCache` nel metodo `ConfigureServices` della classe `Startup`.

Questo metodo accetta opzionalmente un metodo all'interno del quale specifichiamo le opzioni relative all'uso della cache tramite un oggetto di tipo `MemoryDistributedCacheOptions`. Questo tipo espone due proprietà importanti: `SizeLimit`, che indica la quantità massima di dati in cache (in kb), e `ExpirationScanFrequency` che indica ogni quanto tempo effettuare lo scan in memoria.

Esempio 8.18

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddDistributedMemoryCache(o => {
      // Imposta opzioni cache
});
}
```

Se invece vogliamo utilizzare il provider che usa SqlServer come store, dobbiamo usare il metodo `AddDistributedSqlServerCache`, che opzionalmente accetta un metodo per configurare i parametri della cache attraverso un oggetto di tipo `SqlServerCacheOptions`. Questo tipo ha tre proprietà fondamentali: `ConnectionString`, che rappresenta la stringa di connessione al database, `TableName` e `SchemaName` che, rispettivamente, rappresentano il nome della tabella e il relativo schema. L'utilizzo di questo metodo è visibile nel prossimo esempio.

Esempio 8.19

```
services.AddDistributedSqlServerCache(o =>
{
   o.ConnectionString = "connection string";
   o.TableName = "DistributedCache";
   o.SchemaName = "dbo";
});
```

Prima di poter utilizzare il provider per SqlServer, dobbiamo creare nel database la tabella che mantiene i dati. Esiste una riga di comando che copre questa esigenza ed è visibile nell'Esempio 8.20.

Esempio 8.20

```
dotnet sql-cache create "connection string" dbo DistributedCache
```

Utilizzare Redis come store della cache prevede che nel metodo `Configure` della classe `Startup` utilizziamo il metodo `AddDistributedRedisCache`. Anche questo metodo accetta opzionalmente un metodo per configurare le opzioni, che in questo caso sfruttano il tipo `RedisCacheOptions`. Le proprietà di questo tipo sono solo due: `Configuration`, che rappresenta la stringa di connessione a Redis, e `InstanceName`, che specifica il nome dell'istanza del database di Redis.

Esempio 8.21

```
services.AddDistributedRedisCache(o =>
{
   o.Configuration = "connection string";
   o.InstanceName = "db1";
});
```

A prescindere da quale provider utilizziamo, il nostro modo di lavorare con la distributed cache non cambia, in quanto ci interfacciamo con questo servizio sempre tramite l'interfaccia `IDistributedMemoryCache` iniettabile tramite dependency injection, visto che l'abbiamo appena configurata.

Questa interfaccia espone i metodi `Get`, `Set`, `Remove` e, tramite extension method, `SetString` e `GetString`. Ognuno di questi metodi ha anche la sua controparte asincrona, così da permetterne l'utilizzo asincrono.

Il metodo `Set/SetAsync` salva un elemento in cache e permette di impostarne opzionalmente alcune proprietà. Questo metodo accetta in input la chiave e il valore serializzato in un array di byte. Questo formato è necessario, in quanto, se usiamo un provider esterno, i dati devono essere serializzati per poter essere inviati via rete (come abbiamo visto quando abbiamo discusso la sessione). Il metodo `SetString/SetStringAsync` accetta il valore come stringa e poi lo serializza come array di byte prima di inviarlo allo store. Questo metodo torna utile se vogliamo memorizzare non solo stringhe ma anche oggetti complessi. Possiamo infatti serializzare l'oggetto in JSON e poi usare questo metodo per inviarlo alla cache utilizzando un codice simile a quello che abbiamo visto nell'Esempio 8.7.

Per ogni oggetto che aggiungiamo in cache possiamo impostare sia l'absolute expiration sia la sliding expiration, come abbiamo visto per la local cache. Nel prossimo esempio vediamo come utilizzare i metodi appena esaminati.

Esempio 8.22

```
public async Task<IActionResult> Cache(
   [FromServices] IDistributedCache cache)
{
   // elemento senza opzioni
   await cache.SetAsync("key", Serialize("value"));
   // elemento che scade dopo un'ora
   await cache.SetAsync("key", Serialize("value"),
      new DistributedCacheEntryOptions
      {
         AbsoluteExpiration = DateTime.Now.AddHours(1)
      });
   // elemento che scade dopo 10 minuti dall'ultimo accesso
 await cache.SetAsync("key", Serialize("value"),
      new DistributedCacheEntryOptions
      {
         SlidingExpiration = TimeSpan.FromMinutes(10)
      });


   // elemento inserito usando SetString
   await cache.SetStringAsync("key", "value");
   // elemento con un oggetto inserito usando SetString
   await cache.SetStringAsync("key", SerializeJson(obj));
}
```

Il metodo `Get/GetAsync` recupera un valore dalla cache data la sua chiave. Il valore è un array di byte, quindi dobbiamo preoccuparci di deserializzarlo nel tipo reale. Il metodo `GetString/ GetStringAsync` ha lo stesso scopo e parametri del metodo `Get/GetAsync` con la differenza che questo restituisce il valore come stringa. Questo metodo torna utile se vogliamo leggere gli oggetti serializzati in formato JSON.

Esempio 8.23

```
// recupera valore come array di byte
byte[] value = await cache.GetAsync("key");

// recupera valore come stringa
string value = await cache.GetStringAsync("key");

// recupera valore serializzato in json e lo deserializza in un tipo
var value = DeserializeJson<MyClass>(
await cache.GetStringAsync("key"));
```

L'ultimo metodo da analizzare è `Remove/RemoveAsync`, che elimina un elemento dalla cache in base alla chiave che viene passata come parametro di input.

Esempio 8.24

```
await cache.RemoveAsync("key");
```

L'utilizzo della distributed cache non presenta particolari difficoltà, quindi passiamo ora al prossimo argomento, ovvero come mettere in cache porzioni di HTML attraverso appositi tag helper.

HTML caching

Mettere in cache i dati può sicuramente velocizzare le performance di un'applicazione, ma questi dati devono poi essere manipolati e trasformati in HTML. Possiamo velocizzare ulteriormente le performance di un'applicazione se invece che mettere in cache i dati mettiamo in cache il codice HTML generato da questi.

Questa funzionalità è offerta da due tag helper: `cache` e `distributed-cache` che, rispettivamente, usano la local cache e la distributed cache per immagazzinare il codice HTML al loro interno. Questi tag helper condividono gran parte dell'infrastruttura e offrono le stesse opportunità con una sola eccezione: `cache` calcola in automatico la chiave in cache mentre, in `distributed-cache`, questo compito è demandato a noi. L'Esempio 8.25 mostra il codice necessario per usare questi tag helper.

Esempio 8.25

```
<distributed-cache name="cache-key-1">
   Time: @DateTime.Now
</distributed-cache>

<cache>
   Time: @DateTime.Now
<cache>
```

Il frammento di HTML che memorizziamo può contenere informazioni che devono variare al verificarsi di determinate condizioni o informazioni, di cui si devono creare più copie a seconda dei casi. Per esempio, il codice HTML potrebbe contenere il nome dell'utente e quindi dobbiamo memorizzare una versione per ogni utente o, ancora, potremmo dover stabilire una scadenza oltre la quale il frammento HTML va eliminato dalla cache. Queste e altre casistiche sono contemplate dal tag helper, che mette a disposizione vari attributi per gestirle.

Questi attributi sono visibili nella Tabella 8.2.

Tabella 8.2 – Gli attributi dei tag helper di cache.

Attributo	Descrizione
`enabled`	Specifica se salvare in cache il contenuto del tag
`expires-on`	Specifica l'absolute expiration del contenuto
`expires-after`	Specifica dopo quanto tempo il contenuto deve scadere
`expires-sliding`	Specifica la sliding expiration del contenuto
`vary-by-header`	Specifica se creare diverse copie del contenuto in base al valore di una o più intestazioni HTTP (separate da virgola)
`vary-by-query`	Specifica se creare diverse copie del contenuto in base al valore di una o più chiavi in querystring
`vary-by-route`	Specifica se creare diverse copie del contenuto in base ai valori di routing
`vary-by-cookie`	Specifica se creare diverse copie del contenuto in base ai valori di uno o più cookie
`vary-by-user`	Specifica se creare diverse copie del contenuto in base all'utente loggato
`vary-by`	Specifica se creare diverse copie del contenuto in base al valore dell'oggetto passato in input
`priority`	Specifica la priorità per la rimozione dalla cache
`name`	Specifica il nome della chiave di cache che contiene il contenuto. Usato solo per distributed-cache

L'utilizzo di questi attributi è abbastanza semplice, come possiamo vedere nel prossimo esempio.

Esempio 8.26

```
<cache
   expires-on="@new DateTime(2018, 5, 6, 18, 0, 0)"
   expires-sliding="@TimeSpan.FromSeconds(60)"
   vary-by-user="true"
   vary-by-header="accept-language">
      Content: @DateTime.Now - @User.Identity.Name
</cache>
```

In questo codice, impostiamo una data di scadenza assoluta con `expires-on` ma anche una sliding expiration con `expires-sliding`. Nella cache viene creata una copia per ogni utente loggato e per ogni valore dell'header `accept-language`, grazie a `vary-by-user` e `vary-by-header`.

Memorizzare dati o frammenti di HTML in cache può aumentare esponenzialmente le performance di un'applicazione, ma possiamo ulteriormente migliorarle se utilizziamo il response caching.

Response caching

Quando il client invia una richiesta al server, questa passa attraverso diversi intermediari, ognuno dei quali può avere capacità di caching e può quindi rispondere alle richieste senza farle arrivare al server. Non solo un proxy ma anche il client stesso può utilizzare la propria cache per servire pagine già memorizzate senza doverle richiedere al server. Per decidere se mettere in cache o no il risultato di una determinata richiesta, bisogna seguire le specifiche del protocollo HTTP, che prevedono degli specifici header HTTP in base ai quali i vari attori coinvolti nella richiesta possono aggiungere alla cache la risposta data dal server.

Il response caching è la tecnica con la quale ASP.NET permette di impostare e gestire gli header HTTP responsabili della gestione del caching da parte di tutti gli attori di una richiesta.

Il response caching va abilitato in ASP.NET aggiungendo il suo middleware e i relativi servizi, rispettivamente invocando i metodi `UseResponseCaching` e `AddResponseCaching` in fase di configurazione.

Per sfruttare il response caching, dobbiamo applicare l'attributo `ResponseCacheAttribute` alle action o ai controller e specificare le varie opzioni di caching. Se applichiamo l'attributo al controller, tutte le action del controller erediteranno questo attributo. Se, invece, lo applichiamo a una singola action, solo questa verrà abilitata al response caching. Se applichiamo l'attributo sul controller e poi su una action, quello applicato sulla action ha la priorità.

La proprietà più importante di `ResponseCacheAttribute` è `Duration`, che permette di specificare, in secondi, per quanto tempo un contenuto deve essere messo in cache prima che venga eliminato e richiesto al server.

Un'altra proprietà importante è `Location`, che è un enum di tipo `ResponseCacheLocation`, che specifica quali attori possono utilizzare la cache. Il valore `None` disabilita la cache, il valore `Client` abilita la cache solo per il client mentre `Any` abilita la cache per tutti gli attori.

`VaryByHeader` e `VaryByQueryKeys` specificano se creare più copie di cache in base ai valori degli header specificati dalla prima proprietà o in base alle chiavi di querystring specificate dalla seconda. L'ultima proprietà interessante è `NoStore` che, se impostata a `true`, ignora tutte le altre proprietà e impedisce a tutti gli attori di utilizzare la cache.

Nell'Esempio 8.27 vediamo alcuni esempi di utilizzo dell'attributo `ResponseCacheAttribute`.

Esempio 8.27

```
// Mette in cache per 10 secondi
[ResponseCache(Duration = 10)]

// Mette in cache per 10 secondi creando una copia per ogni valore del header
Accept-Lang
[ResponseCache(Duration = 10, VaryByHeader = "Accept-Language")]

// Mette in cache solo sul client per 5 secondi
[ResponseCache(Duration = 5, Location = ResponseCacheLocation.Client)]

// Disabilita la cache
[ResponseCache(NoStore = true)]
```

Ogni action e ogni controller possono avere l'attributo impostato per specificare come mettere in cache. Inoltre, le proprietà dell'attributo molto spesso vengono impostate tutte nello stesso modo, con la conseguenza di avere molto codice duplicato. Per mitigare il problema, possiamo definire in configurazione dei profili di caching, identificabili tramite chiave, e poi sfruttare la proprietà `CacheProfileName` di `ResponseCacheAttribute` impostandola al valore di una delle chiavi, come mostrato nell'Esempio 8.28.

Esempio 8.28

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc(o =>
   {
      o.CacheProfiles.Add("Default",
        new CacheProfile() { Duration = 60 });
   });
}


[ResponseCache(CacheProfileName = "Default")]
public IActionResult CacheableAction(string id)
{
   ...
}
```

L'utilizzo dei profili di cache garantisce un miglior riutilizzo della funzionalità e una facile modifica dei profili.

Conclusioni

In questo capitolo abbiamo analizzato tutte le caratteristiche della gestione dello stato offerte da ASP.NET. La gestione dello stato è un punto fondamentale quando si progetta un'applicazione web, in quanto la scelta di un meccanismo rispetto a un altro può influenzare in maniera decisiva la semplicità del codice, le prestazioni e l'utilizzo di risorse.

In molti casi la scelta tra una tecnica e l'altra non dipende solo dalla semplicità di sviluppo ma anche dalle performance che offre e dai requisiti di sicurezza che dobbiamo rispettare. L'unica cosa che possiamo fare è quella di testare le soluzioni caso per caso, in modo da poter scegliere la via più adatta. Nel prossimo capitolo cambieremo decisamente argomento e andremo ad affrontare un tema fondamentale per qualunque applicazione web: l'accesso ai dati con ADO.NET.

9
Accedere ai dati con ADO.NET

Persistere un dato in sessione o in un cookie o in cache (o altro ancora) è solo una parte della gestione dei dati, cioè quella legata a dati temporanei. La parte più importante della gestione dei dati è quella che li immagazzina all'interno di uno storage persistente come un database.

La tecnologia che .NET Core mette a disposizione per accedere ai dati è ADO.NET. ADO.NET è composto da classi che lavorano a basso livello con il database astraendone le complessità e permettendo a noi di scrivere solamente il codice di business rilevante per la nostra applicazione. Oltre a queste classi, ADO.NET ne mette a disposizione anche altre per manipolare in locale i dati letti dal database per poi aggiornarli sullo stesso database in un momento successivo.

L'utilizzo di ADO.NET assicura l'uniformità della tecnologia per l'accesso ai dati: un sistema di tipi comune, i modelli di progettazione e le convenzioni di denominazione vengono infatti condivisi da tutti i componenti. Inoltre, ADO.NET fornisce un'architettura estendibile che offre una base comune sulla quale creare provider per ogni tipologia di database (SqlServer, Oracle, MySql e così via).

Sulla base di ADO.NET, Microsoft ha costruito un O/RM denominato Entity Framework Core (che approfondiremo nel prossimo capitolo). Questo semplifica notevolmente lo sviluppo del codice di accesso ai dati offrendo funzionalità come la trasformazione dei dati dal database in oggetti, il tracking delle modifiche fatte agli oggetti e la loro persistenza, il supporto a LINQ e molto altro ancora.

Architettura di ADO.NET

Come abbiamo detto, ADO.NET fornisce un sistema di tipi comune e un'architettura estendibile che permette di creare provider (data provider d'ora in poi) per connettersi a qualunque tipo di database (SqlServer, SQLite, Oracle, MySql e così via). Per raggiungere questo scopo, ADO.NET è composto da una serie di namespace che raccolgono le diverse classi per l'accesso ai dati in funzione del loro scopo e della loro implementazione:

- il namespace `System.Data.Common` include le classi base che ogni data provider deve implementare per dialogare con lo specifico database;
- i data provider contengono classi che ereditano da quelle del namespace `System.Data. Common` e che implementano il codice necessario per connettersi a uno specifico database. Generalmente, ogni data provider ha un proprio namespace. Un tipico esempio è il data provider per SQL Server, già presente nel .NET Core, che si trova nel namespace `System. Data.SqlClient`;
- il namespace `System.Data.SqlTypes` contiene le classi che rappresentano i tipi di dati utilizzati in ambito SQL;
- il namespace `System.Data` racchiude le classi di uso generale, indipendenti dalla tipologia di sorgente dati (database, file XML, file JSON e così via), che permettono di lavorare con i dati in locale.

Con quest'architettura ogni gruppo di classi ha un suo specifico ruolo. Vediamo nel dettaglio cosa contengono i namespace menzionati partendo dal primo.

Il namespace System.Data.Common

In `System.Data.Common` si gettano le basi per un modello di programmazione comune, per quanto riguarda l'interfacciamento con il database, in quanto tutti i data provider devono ereditare da queste classi. Le classi più importanti di questo namespace sono:

- `DbConnection`: permette di stabilire una connessione con il database;
- `DbCommand`: permette di eseguire comandi sul database sia per leggere che per scrivere dati;
- `DbDataReader`: permette di leggere i dati recuperati tramite una query eseguita con il `DbCommand`;
- `DbTransaction`: permette di aprire una transazione con il database garantendo che le modifiche ai dati siano effettive solamente se ogni comando va a buon fine. In caso di errori le modifiche vengono annullate;
- `DbParameter`: permette di specificare un parametro per il comando SQL;
- `DbConnectionStringBuilder`: permette di costruire programmaticamente una stringa di connessione al database;

- `DbDataAdapter`: incapsula la logica di connessione, transazione e comandi permettendo di eseguire query di lettura e scrittura in maniera molto semplice integrandosi anche con le classi in `System.Data`. Con i moderni paradigmi di programmazione a oggetti, questa classe è divenuta obsoleta ed è presente in .NET Standard esclusivamente per retrocompatibilità con lo scopo di favorire il porting di applicazioni legacy da .NET a .NET Core. Per questo motivo non approfondiremo l'utilizzo di questa classe e delle sue derivate nei data provider ma ne parleremo solo superficialmente.

Queste classi sono astratte e rappresentano la base per il colloquio con il database. Questa lista non è completa in quanto il numero completo di classi presenti nel namespace è 33. Tuttavia elencarle tutte sarebbe inutile in quanto questa lista comprende le classi che vengono usate nel 99% dei casi quindi è più che sufficiente per capire come è fatta la base di un data provider. Vediamo ora come vengono ereditate e implementate queste classi dai data provider.

Data provider

Le classi dei data provider sono quelle che fisicamente usiamo per interagire con il database: connetterci, eseguire query, scrivere dati e così via. Queste classi non sono altro che un'estensione di quelle viste nel precedente paragrafo.

.NET Core contiene già un data provider per SqlServer, contenuto nel namespace `System.Data. SqlClient`, quindi prendiamo questo come esempio per descrivere le classi:

- `SqlConnection`: eredita da `DbConnection` e permette di stabilire una connessione a SqlServer;
- `SqlCommand`: eredita da `DbCommand` e permette di eseguire comandi su SqlServer;
- `SqlDataReader`: eredita da `DbDataReader` e permette di leggere i dati recuperati tramite una query eseguita con il `SqlCommand`;
- `SqlTransaction`: eredita da `DbTransaction` e permette di aprire una transazione su SqlServer.
- `SqlParameter`: eredita da `DbParameter` e permette di specificare un parametro per il comando SQL lanciato su SqlServer.
- `SqlConnectionStringBuilder`: eredita da `DbConnectionStringBuilder` e permette di costruire programmaticamente una stringa di connessione al database SqlServer;
- `SqlDataAdapter`: eredita da `DbDataAdapter` e ne incapsula le logiche specializzandole per SqlServer;

Quello che appare evidente da questa lista, è che per creare un data provider tutto quello che bisogna conoscere sono le classi base da cui ereditare e le competenze tecnologiche del database verso il quale ci si vuole interfacciare. Non è un caso che i vari vendor di database abbiano già creato e distribuito, tramite pacchetto NuGet, il loro data provider per .NET Core.

Infatti, oltre ai provider già inclusi in .NET Core (SqlServer e SQLite), esistono provider forniti da terze parti per altri database come MySql, PostgreSQL e Firebird per nominare quelli più comuni. Al momento della stesura di questo libro, Oracle ha rilasciato su NuGet il provider ufficiale per .NET Core, ma questo è ancora in versione beta, e quindi il suo utilizzo in scenari di produzione non è ancora consigliato. All'indirizzo `http://aspit.co/r9` potete seguire i rilasci di Oracle per il mondo .NET e .NET Core.

> *Oledb e Odbc sono stati portati in .NET Core 2.0 ma non fanno parte di .NET Standard 2.0, quindi possiamo utilizzarli solo se stiamo creando un'applicazione basata sul primo.*

Una volta capito come lavorare fisicamente con il database è importante capire come mappare i dati verso i tipi analoghi .NET.

Il namespace System.Data.SqlTypes

Una stringa in .NET Core è differente da una stringa nel database in quanto la loro rappresentazione interna cambia. Tuttavia, in .NET Core è molto più comodo usare la rappresentazione nativa delle stringhe piuttosto che quella del database. Per questo motivo, sono state create nel namespace `System.Data.SqlTypes` classi che agiscono da mapper tra i tipi .NET e i tipi del database. In questo modo noi possiamo lavorare usando i tipi nativi .NET Core mentre quando i data provider interagiscono con il database usano queste classi. All'interno di `System.Data.SqlTypes` abbiamo classi come `SqlString`, `SqlInt32`, `SqlByte`, `SqlBinary`, `SqlBoolean`, `SqlDateTime` e altre ancora. Il nome delle classi è abbastanza esplicativo quindi non occorre spiegare quali tipi queste mappino.

Il namespace System.Data

Le classi in `System.Data` permettono di accedere ai dati recuperati dal database, di manipolarli ed eventualmente persisterli sul database. Le classi principali di questo namespace sono quattro:

- `DataTable`: classe che contiene la lista di record recuperati da una query sul database;
- `DataRow`: classe che contiene un singolo record letto dal database. Un oggetto `DataTable` contiene una lista di `DataRow`;
- `DataColumn`: classe che rappresenta una colonna all'interno di una riga. Un oggetto `DataRow` contiene una lista di `DataColumn`;
- `DataSet`: classe che rappresenta un contenitore di `DataTable`.

Queste classi ricalcano la struttura di un database. Un oggetto `DataSet` è come un database, un oggetto `DataTable` è paragonabile a una tabella con tante righe (oggetti `DataRow`) ognuna delle quali composta da tante colonne (oggetti `DataColumn`). Grazie a questa struttura, queste classi rendono il modello di programmazione molto familiare con il database.

L'immagine 9.1 mostra l'architettura di ADO.NET

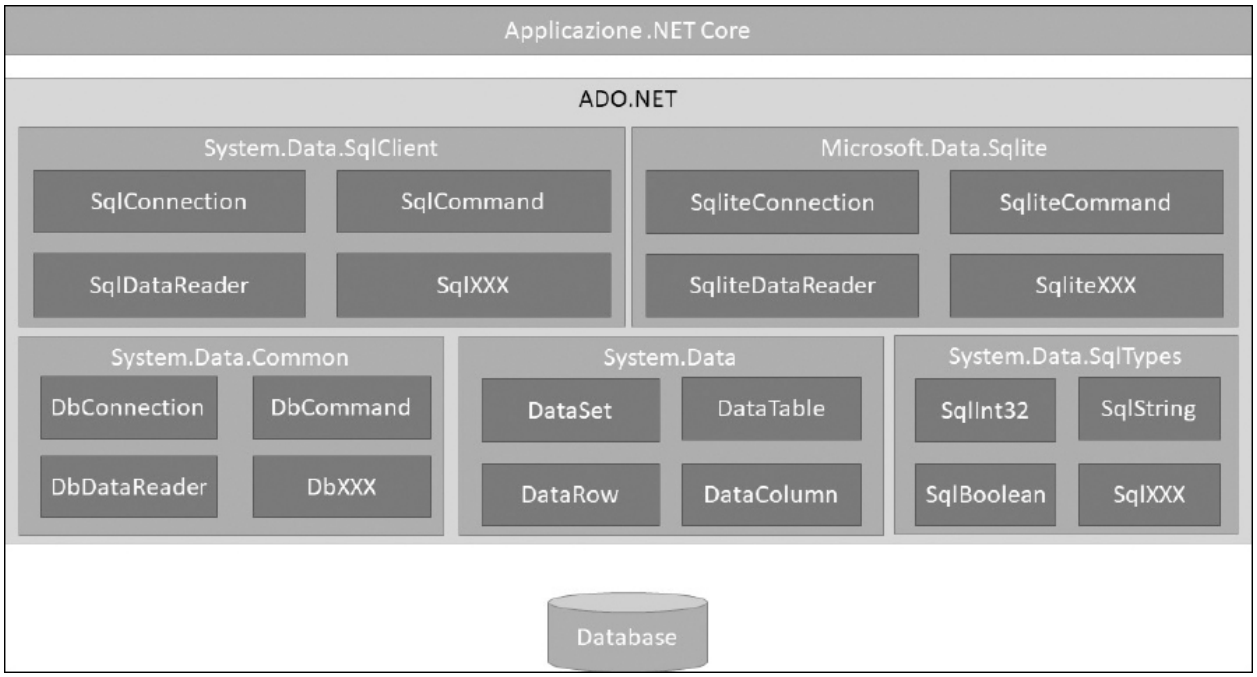


Figura 9.1 – Architettura di ADO.NET.

Conoscere l'architettura di ADO.NET è importante, perché ci fornisce le basi per scrivere codice che accede ai dati. Nella prossima sezione metteremo a frutto queste conoscenze scrivendo questo codice.

Lavorare con ADO.NET

Il primo passo per lavorare con qualunque database è stabilire una connessione. Infatti, per comunicare con un database al fine di eseguire comandi di lettura e di aggiornamento, è sempre necessario instaurare un certo tipo di collegamento, che dipende inevitabilmente dal tipo di database. Nel caso dei database relazionali, l'attivazione di una connessione fisica al server è propedeutica all'inoltro di qualsiasi comando SQL per la lettura o la modifica dei dati contenuti nelle tabelle. Vediamo come stabilire una connessione con il database utilizzando SqlServer.

Stabilire una connessione

Come detto in precedenza, per stabilire una connessione con il database bisogna usare la classe del data provider che eredita da `DbConnection`. Questa classe espone la proprietà `ConnectionString` che rappresenta una stringa tramite la quale specifichiamo i parametri di connessione al database. La stringa di connessione per una particolare istanza può essere specificata sia tramite la proprietà appena menzionata, sia in fase di creazione tramite un costruttore che accetta la stringa di connessione come parametro.

Una volta impostata la stringa di connessione, apriamo e chiudiamo la connessione utilizzando rispettivamente i metodi `Open` (o la sua controparte asincrona `OpenAsync`) e `Close`. L'Esempio 9.1 mostra come aprire e chiudere una connessione verso un database SqlServer, definendo la stringa di connessione tramite il costruttore parametrico della classe `SqlConnection`. L'utilizzo del blocco di gestione delle eccezioni permette di intercettare in modo appropriato gli eventuali errori, derivanti, per esempio, da un'errata

configurazione delle credenziali dell'utente oppure da un problema di connessione al server. Una volta aperta, la connessione a una sorgente dati va sempre chiusa in modo esplicito, per evitare sprechi di risorse.

Esempio 9.1

```
// Stringa di connessione
var connectionString = "Server=localhost;Database=Northwind;
   User ID=appUser; Password=p@$$w0rd";
// Creazione dell'istanza di SqlConnection
var conn = new SqlConnection(connectionString);

try
{
   // Apertura della connessione
   await conn.OpenAsync();
   // ...
}
catch(SqlException ex)
{
   // Gestione dell'eccezione
}
finally
{
   // Chiusura della connessione
   if (conn.State == ConnectionState.Open)
      conn.Close();
}
```

Dal momento che la classe `DbConnection` implementa l'interfaccia `IDisposable`, il codice precedente può essere scritto in modo più compatto, sfruttando il costrutto `using`.

Esempio 9.2

```
var connectionString = "...";

using(var connection = new SqlConnection(connectionString))
{
   try
   {
      await connection.OpenAsync();
      // ...
   }
   catch(SqlException ex)
   {
      // Gestione dell'eccezione
   }
}
```

Il codice riportato nell'Esempio 9.2 è del tutto equivalente a quello mostrato nell'Esempio 9.1. La connessione viene chiusa in modo trasparente al termine del blocco `using`, mediante una chiamata implicita del metodo `Dispose` sia in caso di successo sia in caso di errore.

La stringa di connessione è composta da una serie di coppie nome/valore separate dal carattere ";" (punto-e-virgola), dove il nome corrisponde a una parola chiave. Le principali parole chiave sono elencate qui di seguito:

- `Data Source` (equivalente a `Server`) specifica il percorso dove risiede la sorgente dati;
- `Database` (equivalente a `Initial Catalog`) identifica il database predefinito;
- `User ID` (equivalente a `Uid`) identifica il nome dell'utente nel caso in cui sia necessario specificare le credenziali di accesso (per esempio, autenticazione SqlServer);
- `Password` (equivalente a `Pwd`) identifica la password dell'utente nel caso in cui sia necessario specificare le credenziali di accesso;
- `Integrated Security` (equivalente a `Trusted_Connection`) permette di abilitare l'autenticazione Windows (autenticazione integrata). In questo caso, le credenziali dell'utente possono essere omesse;

Oltre a queste parole chiave, ne esistono altre di cui alcune valide per ogni database e altre specifiche per tipologia di database. Elencare tutte le parole chiave è un'operazione che esula dagli scopi di questo testo in quanto richiederebbero molto spazio e la conoscenza di ogni parola chiave specifica per database.

Negli esempi analizzati in precedenza, abbiamo specificato la stringa di connessione direttamente nel codice, ma questa pratica non è utilizzabile in un contesto reale dove la stringa di connessione deve provenire dalla configurazione. Nel file di configurazione dell'applicazione, possiamo specificare, nella sezione `ConnectionStrings`, l'elenco delle stringhe di connessione disponibili per l'applicazione. Ogni stringa di connessione è specificata da una coppia chiave/valore dove la chiave è un identificativo della stringa e il valore è la stringa di connessione (Esempio 9.3).

Esempio 9.3

```
{
   "ConnectionStrings": {
      "SQL": "Server=dbserver;Database=northwind;Integrated security=true"
   }
}
```

Le stringhe di connessione sono accessibili tramite il metodo `GetConnectionString` dell'interfaccia `IConfiguration`. Questo metodo accetta in input il nome della chiave relativa alla stringa di connessione e restituisce la stringa. Nell'Esempio 9.4 iniettiamo un oggetto di tipo `IConfiguration` nel costruttore della classe e poi usiamo `GetConnectionString` per recuperare la stringa di connessione con chiave SQL.

Esempio 9.4

```
public class CustomerService
{
   private readonly string connectionString;
   public CustomerService(IConfiguration configuration)
   {
      connectionString = configuration.GetConnectionString("SQL");
   }
}
```

Oltre a essere recuperata dal file di configurazione, una stringa di connessione può essere costruita in modo programmatico sfruttando l'implementazione della classe `DbConnectionStringBuilder` che ogni data provider fornisce.

La classe `DbConnectionStringBuilder` permette di assemblare la stringa di connessione, fornendo un controllo intrinseco sul formato e sulla validità dei vari parametri. A ciascun parametro della stringa di connessione corrisponde una proprietà dell'oggetto builder, che deve essere impostata in modo opportuno. Una volta valorizzati i parametri necessari, la stringa finale è accessibile tramite la proprietà `ConnectionString`. L'Esempio 9.5 mostra la costruzione di una stringa di connessione per accedere a un database SqlServer tramite `SqlConnectionStringBuilder`.

Esempio 9.5

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = serverName;
builder.InitialCatalog = database;
builder.IntegratedSecurity = true;
var connectionString = builder.ConnectionString;
```

Nel caso in cui le parti della stringa di connessione derivino da un input da parte dell'utente, l'utilizzo dell'oggetto builder migliora decisamente la sicurezza, riducendo in modo significativo il rischio di attacchi basati su input malevoli.

Stabilire la connessione con il database è il primo passo per dialogare, il secondo consiste nell'esecuzione di comandi sul database a cui si è connessi.

Esecuzione di un comando

Una volta che la connessione è stata attivata, possiamo inviare comandi alla sorgente dati per la lettura e per la scrittura. A tale scopo, il modello a oggetti di ADO.NET fornisce il tipo base astratto `DbCommand`, che include una serie di membri comuni a tutte le implementazioni, presenti nei vari data provider. La Tabella 9.1 riporta le proprietà e i metodi di uso più frequente.

Tabella 9.1 – Membri principali della classe DbCommand.

Proprietà o metodo	Descrizione
`CommandText`	Proprietà che imposta lo statement SQL o il nome della stored procedure da eseguire.
`CommandType`	Proprietà di tipo CommandType (enumerazione) che definisce la tipologia del comando. I valori possibili sono: Text (default), StoredProcedure, TableDirect.

Connection	Proprietà che associa una connessione al comando.
Parameters	Proprietà che rappresenta la collezione dei parametri di input e output utilizzati dal comando.
Transaction	Proprietà che permette di definire a quale transazione associata alla connessione il comando appartiene.
ExecuteNonQuery ExecuteNonQueryAsync	Metodi per l'esecuzione di un comando diverso da una query. Il primo viene eseguito in modalità sincrona e ritorna il numero di righe interessate. Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<int> che espone il numero di righe interessate.
ExecuteReader ExecuteReaderAsync	Metodi per l'esecuzione di una query. Il primo viene eseguito in modalità sincrona e ritorna un cursore di tipo forward-only e readonly, contenente il risultato. Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<DbDatareader> che espone il cursore.
ExecuteScalar ExecuteScalarAsync	Metodi per l'esecuzione di una query. Il primo viene eseguito in modalità sincrona e ritorna il valore presente nella prima colonna della prima riga del risultato (di tipo object). Gli altri dati vengono ignorati. Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<object> che espone il valore della prima colonna della prima riga.

Come possiamo vedere nella Tabella 9.1, i metodi per l'esecuzione di un comando sono tre (in realtà sei se consideriamo che ci sono sia le versioni sincrone che quelle asincrone). Ciascuna funzione restituisce un valore di ritorno differente, a testimonianza del fatto che i metodi sono stati concepiti per un utilizzo specifico. ExecuteNonQuery ed ExecuteNonQueryAsync di invocare un comando di inserimento (INSERT), aggiornamento (UPDATE) o cancellazione (DELETE) e ritornano il numero di righe interessate. ExecuteReader ed ExecuteReaderAsync consentono di eseguire interrogazioni sui dati (SELECT), restituendo uno o più resultset a seconda del numero di query che inviamo. ExecuteScalar ed ExecuteScalarAsync permettono infine di recuperare un valore singolo da una query e si rivelano particolarmente efficaci nel caso di comandi che recuperano valori aggregati come, per esempio, SELECT COUNT, SELECT MAX, SELECT MIN e così via.

L'Esempio 9.6 riporta le tre casistiche d'esecuzione di un comando sincrono e asincrono verso un database SqlServer.

Esempio 9.6

```
//Metodo sincrono
private void Method(SqlConnection connection)
{
   // Aggiornamento
   var cmdUpdate = new SqlCommand("UPDATE Products SET ...", connection);
   int affectedRows = cmdUpdate.ExecuteNonQuery();
   // Query
   var cmdQuery = new SqlCommand("SELECT * FROM Products", connection);
   SqlDataReader reader = cmdQuery.ExecuteReader();
   // Conteggio
   var cmdCount = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
   var count = (int)cmdCount.ExecuteScalar();
}

//Metodo asincrono
private async Task MethodAsync(SqlConnection connection)
{
   // Aggiornamento
   var cmdUpdate = new SqlCommand("UPDATE Products SET ...", connection);
   int affectedRows = await cmdUpdate.ExecuteNonQueryAsync();
   // Query
   var cmdQuery = new SqlCommand("SELECT * FROM Products, connection ");
   SqlDataReader reader = await cmdQuery.ExecuteReaderAsync();
   // Conteggio
   var cmdCount = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
   var count = (int)(await cmdCount.ExecuteScalarAsync());
}
```

Il testo del comando non deve essere necessariamente espresso per esteso. Infatti, DbCommand permette di eseguire anche stored procedure, specificando la tipologia del comando mediante la proprietà CommandType. Tra le opzioni possibili, contenute nell'enumerazione System.Data.CommandType, il valore StoredProcedure consente di specificare l'intenzione di invocare una stored procedure sul database. In tal caso, la proprietà CommandText del comando deve contenere il nome della stored procedure invece del testo SQL formattato esplicitamente.

Per eseguire comandi SQL o stored procedure dotate di valori in ingresso, possiamo usare i parametri. Essi sono istanze delle classi che derivano dal tipo base DbParameter e sono caratterizzati da un nome identificativo, un valore, un tipo, una dimensione e una

direzione (input/output). Ciascun parametro può essere associato a un comando tramite il metodo `Add` della proprietà `Parameters` (Esempio 9.7).

Esempio 9.7

```
var query = new SqlCommand(
   ”SELECT * FROM Orders ” +
   ”WHERE EmployeeID = @EmployeeID ” +
   ”AND OrderDate = @OrderDate ” +
   ”AND ShipCountry = @ShipCountry” +
   ”ORDER BY OrderDate DESC”, connection);
var p1 = new SqlParameter
{
   ParameterName = ”@EmployeeID”;
   DbType = DbType.Int32;
   Direction = ParameterDirection.Input;
   Value = 1;
}
var p2 = new SqlParameter {
   ParameterName = ”@OrderDate”;
   DbType = DbType.DateTime;
   Direction = ParameterDirection.Input;
   Value = new DateTime(1996, 8, 7);
}
var p3 = new SqlParameter
{
   ParameterName = ”@ShipCountry”;
   DbType = DbType.String;
   Direction = ParameterDirection.Input;
   Value = ”Italy”;
}
query.Parameters.Add(p1);
query.Parameters.Add(p2);
query.Parameters.Add(p3);
```

Un approccio alternativo all’uso dei parametri (purtroppo ancora diffuso fra gli sviluppatori) consiste nell’utilizzare la concatenazione di stringhe, allo scopo di comporre il testo del comando SQL includendo i valori in ingresso.

Anche se, come soluzione, può sembrare equivalente a quella basata su parametri e anche più veloce da implementare, l’uso della concatenazione rappresenta un approccio sbagliato e, quindi, assolutamente da evitare per motivi di sicurezza applicativa. Infatti, la semplice concatenazione di stringhe non permette di controllare se i valori in ingresso sono formattati correttamente. Pertanto, la concatenazione consente l’iniezione di codice maligno all’interno del testo del comando SQL, con conseguenze che, nella maggior parte dei casi, si possono rivelare disastrose. Molti attacchi alle applicazioni sfruttano proprio l’uso della concatenazione di stringhe nella formattazione del codice SQL per poter eseguire comandi non previsti dallo sviluppatore e per modificare i dati contenuti nelle tabelle del database.

L’approccio basato su parametri garantisce il giusto livello di sicurezza, dal momento che non consente in alcun modo l’iniezione di codice SQL. Per questo motivo, questa soluzione è sempre da preferire. Essa permette di proteggersi dagli attacchi di tipo SQL-Injection (iniezione di codice SQL maligno) e garantisce il controllo semantico dei dati in ingresso. Infatti, grazie all’uso dei parametri, la formattazione di date e numeri oppure la codifica dei caratteri speciali, come l’apice singolo, vengono eseguite in modo trasparente, indipendentemente dalle impostazioni internazionali di sistema e dai settaggi del database (come nel caso del secondo parametro dell’Esempio 9.7).

Possiamo usare i parametri con tutti i data data provider. La regola generale prevede di utilizzare il nome del parametro preceduto dal carattere “@” come marcatore all’interno del testo del comando (Esempio 9.7).

Quando inviamo più comandi di scrittura, possiamo inglobarli in una transazione, per garantire che tutti siano eseguiti o annullati senza lasciare dati inconsistenti sul database. Vediamo ora come gestire le transazioni.

Scrivere dati in transazione

Quando inviamo un commando di scrittura, il database esegue l’aggiornamento dei dati e ritorna il numero di righe modificate. Tuttavia, spesso capita di dover eseguire più comandi e che questi comandi debbano essere eseguiti in transazione, per garantire che vengano tutti confermati o annullati.

I data provider supportano le transazioni attraverso una classe che eredita da `DbTransaction`. Questa classe viene istanziata sfruttando il metodo `BeginTransaction` della classe che eredita da `DbConnection`. `BeginTransaction` ritorna un oggetto `DbTransaction` sul quale poi possiamo invocare i metodi `Commit` e `Rollback` che, rispettivamente, confermano e annullano i

comandi inviati nel contesto della transazione. Nell'Esempio 9.8 vediamo come creare una transazione, inviare comandi e, infine, confermare o annullare la transazione.

Esempio 9.8

```
using (var connection = new SqlConnection(connectionString))
{
   connection.Open();
   using (SqlTransaction transaction = connection.BeginTransaction())
   {
      try
      {
         // inserisce un ordine
         // inserisce i dettagli
         // aggiorna il magazzino
         transaction.Commit();
      }
      catch (SqlException ex)
      {
         transaction.Rollback();
         // Gestione dell'eccezione
      }
   }
}
```

In questo esempio, prima apriamo la connessione e poi iniziamo la transazione; successivamente inviamo i comandi al database e, se non ci sono errori, eseguiamo il commit della transazione. Se invece ci sono errori, il controllo del codice passa per il blocco `catch`, all'interno del quale facciamo il rollback della transazione che annulla tutti i comandi inviati nel contesto della transazione stessa. Infine, alla chiusura dei blocchi `using`, viene fatto il dispose della transazione e della connessione. Questa modalità di gestione delle transazioni è sicuramente semplice quando abbiamo un solo metodo che manipola i dati ma, quando abbiamo più metodi che devono scrivere in transazione, dobbiamo fare in modo che questa sia accessibile, passandola a ogni metodo come parametro o rendendola disponibile in una classe di contesto accessibile dai metodi. Il codice diventa sicuramente più complicato da scrivere anche in virtù del fatto che la transazione potrebbe non essere sempre attiva, a seconda della nostra logica di business.

Per semplificare gli scenari transazionali, ADO.NET mette a disposizione le classi del namespace `System.Transaction` la cui classe principale è `TransactionScope`. Grazie a questa classe, possiamo creare una transazione esplicita, che viene memorizzata nelle informazioni del thread (e che sopravvive anche ai cambi di thread qualora usassimo `async/await`) e che viene automaticamente utilizzata dagli oggetti di ADO.NET. In questo modo, nel nostro codice tutto quello che dobbiamo fare è iniziare la transazione e invocarne il commit quando il nostro codice viene eseguito con successo.

Esempio 9.9

```
public void CreateOrder(Order order)
{
   using (var scope = new TransactionScope())
   {
      SaveOrder(order);
      UpdateStock(order);
      scope.Complete();
   }
}

private SaveOrder(Order order)
{
   using (var connection = new SqlConnection(connectionString))
   {
      connection.Open();
      //comandi di salvataggio dell'ordine
   }
}

private UpdateStock(Order order)
{
   using (var connection = new SqlConnection(connectionString))
   {
      connection.Open();
      //comandi di aggiornamento del magazzino prodotti
   }
```

```
}
```

Nel metodo principale di salvataggio, prima creiamo un oggetto `TransactionScope`, così da creare un contesto transazionale. Successivamente chiamiamo i metodi che compiono fisicamente le operazioni sul database e infine invochiamo il metodo `Complete` di `TransactionScope`, che indica che, quando lo scope viene chiuso (in questo caso quando si arriva alla fine del blocco `using`), la transazione deve essere committata. Se non invochiamo il metodo `Complete`, al momento della chiusura la transazione viene annullata, lanciando in automatico un rollback.

I metodi che scrivono sul database sono completamente agnostici rispetto alla transazione. Gli oggetti di ADO.NET sono in grado di capire che si trovano in un contesto transazionale e utilizzano la transazione in automatico. Questo, per la leggibilità del codice, è un grosso passo in avanti rispetto all'utilizzo della classe `DbTransaction` e sue derivate.

Ora che siamo in grado di scrivere dati sul database, cambiamo argomento e vediamo come leggerli.

Lettura del risultato di una query

Come detto, l'esecuzione dei metodi `ExecuteReader` e `ExecuteReaderAsync` comporta la restituzione di un oggetto contenente i risultati dell'interrogazione. Questo oggetto, detto genericamente data reader, è un'istanza di una delle classi che derivano dal tipo astratto `DbDataReader`, contenuto nel namespace `System.Data.Common`. Un data reader rappresenta un cursore client-side di tipo forward-only e read-only, che consente di scorrere e leggere uno o più resultset, generati da un comando associato a una connessione.

Grazie al data reader, possiamo accedere ai dati un record alla volta, utilizzando il metodo `Read` o il suo corrispondente asincrono `ReadAsync`. Dal momento che la funzione ritorna il valore *true* finché tutti i dati non sono stati consumati, può essere usata come condizione d'uscita in un ciclo iterativo di lettura. Chiaramente, il blocco associato al ciclo deve contenere il codice per trattare i dati relativi al record corrente (Esempio 9.10).

La lettura dei campi può essere eseguita in due modi:

- mediante indexer, che permette di recuperare il valore di una colonna nel formato nativo in base all'indice o al nome del campo; in questo caso dobbiamo eseguire un'operazione di casting, in funzione del tipo di destinazione;

- tramite i metodi `GetXXX`, che permettono di leggere i campi in funzione della loro posizione all'interno del record, ritornando direttamente uno specifico tipo di dato (per esempio, `GetInt32` ritorna un intero, `GetDateTime` ritorna una data, `GetString` ritorna una stringa ecc.).

Esempio 9.10

```
using (var cmd = new SqlCommand("SELECT * FROM Products"), connection)
{
   using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
   {
      while(reader.Read())
      {
         // Viene utilizzata la proprietà indexer
         var productID = (int)reader["ProductID"];
         // ProductName è il secondo campo del record
         var productName = reader.GetString(1);
         // ...
      }
   }
}
```

Il data reader alloca risorse che devono poi essere eliminate nel momento in cui finiamo di leggere i dati. Il modo più semplice per deallocare le risorse è invocando il metodo `Close`. Dal momento che la classe `DbDataReader` implementa l'interfaccia `IDisposable`, possiamo usare la sintassi che fa uso del blocco `using` così da invocare automaticamente il metodo `Dispose` che, internamente, chiama il meteodo `Close`.

Un data reader può contenere più di un resultset. Questo avviene quando al comando che ha generato il data reader sono associate più query. In questo caso, il data reader, una volta creato, viene sempre posizionato sul primo resultset. Per spostarsi da un resultset a quello successivo è necessario utilizzare il metodo `NextResult` o la sua controparte asincrona `NextResultAsync`. Il metodo restituisce il valore *false* se non esistono altri resultset da leggere all'interno del data reader corrente.

Esempio 9.11

```
var cmd = new SqlCommand("SELECT * FROM Products; SELECT * FROM Customers",
   connection);
using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
{
   IterateOverProducts(reader);
```

```
    await reader.NextResultAsync();
    IterateOverCustomers(reader);
 }
```

L'uso più comune di un data reader è quello di leggere i dati per poi riversarli in uno o più oggetti. L'iterazione degli elementi di un data reader richiede, però, una connessione aperta. Vediamo ora come leggere i dati senza mantenere una connessione aperta con il database.

Modalità disconnessa in ADO.NET

Oltre alla modalità connessa che, come abbiamo visto, contempla l'utilizzo dei data reader, ADO.NET permette di lavorare sui dati anche in modalità disconnessa, ovvero senza che sia attiva una connessione verso la sorgente dati. Tuttavia, è utile sottolineare che la modalità disconnessa, sebbene sia stata ampiamente adottata dagli sviluppatori nelle primissime versioni del .NET Framework, oggi rappresenta un metodo assolutamente superato e, di conseguenza, sconsigliato per accedere ai dati. L'avvento di tecnologie alternative e più evolute, come Entity Framework, oggetto della prossima sezione, ne hanno decretato inevitabilmente l'obsolescenza. Per completezza, in questo contesto ci limitiamo a riportare un veloce richiamo dei concetti principali, senza entrare nei dettagli.

Per poter lavorare sui dati in modalità disconnessa, ADO.NET include un oggetto particolare, simile a un comando, detto genericamente data adapter, tramite il quale possiamo popolare un oggetto container con le informazioni recuperate dalla sorgente dati. Un container di dati è un oggetto finalizzato a raccogliere, in modo strutturato e ordinato, le informazioni che in esso vengono inserite, fornendo al tempo stesso una serie di funzionalità per il trattamento e la lettura del suo contenuto.

Il data adapter si comporta come tramite, in entrambi i sensi, tra la sorgente dati e il container. Infatti, a differenza del comando dove il resultset viene ritornato sotto forma di cursore read-only, il data adapter sfrutta l'oggetto container come raccoglitore delle informazioni recuperate dalla sorgente dati. In ADO.NET esistono due tipi di container di dati: il `DataSet` e il `DataTable`.

> *Le classi container di ADO.NET non sono elementi specifici di un particolare data provider. Sono altresì dei semplici contenitori, che presentano una serie di funzionalità simili a quelle offerte da un database classico, come l'organizzazione dei dati in tabelle, le relazioni, l'integrità referenziale, i vincoli, l'indicizzazione e così via. Lo scopo dei container è quello di ospitare insiemi di dati, strutturati secondo uno schema specifico, mantenendoli attivi in memoria affinché possano essere letti e modificati in modo semplice e immediato. In particolare, il* `DataSet` *è un oggetto composto da un insieme di tabelle, rappresentate da altrettante istanze della classe* `DataTable` *e da relazioni di tipo* `DataRelation`. *Ciascuna tabella, a sua volta, è composta da righe (classe* `DataRow`*) e colonne (classe* `DataColumn`*) e può includere vincoli di integrità referenziale e di univocità dei dati.*

Il data adapter è una classe che deriva dal tipo base `DbDataAdapter`. Ogni data provider presenta una sua implementazione specifica, ma tutte le specializzazioni includono i membri utili al popolamento e all'aggiornamento di un particolare container di dati. La Tabella 9.2 riporta le proprietà e i metodi principali.

Tabella 9.2 – Membri principali della classe DbDataAdapter.

Proprietà o metodo	Descrizione
`SelectCommand`	Proprietà che permette di impostare il comando di selezione delle informazioni provenienti dalla sorgente dati. Questo comando viene utilizzato dal data adapter durante l'operazione di popolamento del container di dati di destinazione (DataSet o DataTable).
`InsertCommand`	Proprietà che permette di impostare il comando di inserimento di nuovi record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
`UpdateCommand`	Proprietà che permette di impostare il comando di aggiornamento dei record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
`DeleteCommand`	Proprietà che permette di impostare il comando di cancellazione dei record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
`Fill(DataSet)`	Metodo per il popolamento di un DataSet. Il metodo è soggetto a overloading.
`Fill(DataTable)`	Metodo per il popolamento di una DataTable. Il metodo è soggetto a overloading.
`Update(DataSet)`	Metodo per il salvataggio (batch update) del contenuto di un DataSet verso la sorgente dati. Il metodo è soggetto a overloading.
`Update(DataTable)`	Metodo per il salvataggio (batch update) del contenuto di una DataTable verso la sorgente dati. Il metodo è soggetto a overloading.

Dal momento che funge da tramite "da e verso" la sorgente dati, il data adapter include internamente quattro comandi (a cui corrispondono le quattro proprietà in tabella) che vengono invocati per le operazioni di lettura e di salvataggio dei dati (Esempio 9.12). Sia durante l'operazione di popolamento sia durante quella di aggiornamento (detta anche batch update), il data adapter attiva, in modo trasparente, una connessione verso la sorgente dati e invoca i comandi in relazione all'operazione che sta compiendo.

Esempio 9.12

```
var conn = new SqlConnection("...");

// In fase di creazione occorre specificare la connessione
var adapter = new SqlDataAdapter("SELECT * FROM Products", conn);

// Creazione della DataTable
var dt = new DataTable();

// Popolamento della DataTable
adapter.Fill(dt);

// Modifica dei dati contenuti nella DataTable...

// Batch update
adapter.Update(dt);
```

La connessione, associata al comando di selezione all'atto della chiamata del metodo di popolamento `Fill`, deve essere valida, ma non necessariamente aperta. Se la connessione risulta essere chiusa prima della chiamata della funzione di popolamento, essa viene automaticamente aperta e, suc-cessivamente, chiusa dal data adapter in modo del tutto trasparente. Se, invece, la connessione risulta essere aperta prima della chiamata del metodo `Fill`, essa viene lasciata aperta dal data adapter e deve essere chiusa in modo esplicito.

Ricercare dati in un DataSet

Una volta popolato, possiamo effettuare ricerche all'interno di un dataset attraverso LINQ to DataSet. Questi sono extension method aggiunti alla classe `DataTable` che ne trasformano le righe in una collection tipizzata e, successivamente, ne permettono la ricerca nello stile LINQ.

Il metodo che trasforma le righe in una collection ricercabile è `AsEnumerable`. Una volta ottenuta la collection, possiamo usare il metodo `Where` per filtrare i dati. Questo metodo accetta in input una funzione che prende la riga come parametro. Per recuperare i valori dei campi all'interno della riga, possiamo usare l'extension method `Field`, che accetta il tipo del campo, come parametro generico, e il nome. L'Esempio 9.13 mostra come filtrare i clienti del database.

Esempio 9.13

```
var ds = // popola dataset
EnumerableRowCollection<DataRow> enumDt = c.Tables[0].AsEnumerable();
var custs = enumDt.Where(t => t.Field<string>("CustomerID").StartsWith("A");
```

Oltre al metodo `Where`, ci sono anche i metodi `Select` e `OrderBy` che svolgono le medesime funzioni svolte dagli omonimi metodi LINQ di base. Il loro funzionamento è lo stesso visto per il metodo `Where`.

Conclusioni

L'accesso ai dati è probabilmente la parte più importante di ogni applicazione e ADO.NET rappresenta il sottosistema di accesso ai dati all'interno di .NET.

ADO.NET fornisce agli sviluppatori tutti gli strumenti necessari per accedere ai dati, per leggerli e per modificarli anche in un contesto transazionale.

Grazie alla sua struttura a provider, ADO.NET permette di scrivere questo codice in maniera quasi agnostica rispetto al database e permette di creare facilmente specializzazioni di ADO.NET per ogni tipo di database (SqlServer, Oracle, MySql e altro ancora).

Sulla base di ADO.NET, Microsoft ha costruito un O/RM chiamato Entity Framework Core. L'utilizzo di un O/RM rende il nostro codice di accesso ai dati estremamente più pulito e robusto, semplificandone lo sviluppo. Nel prossimo capitolo ci occuperemo di questo argomento.

10
Accedere ai dati con Entity Framework Core

Nel capitolo precedente abbiamo illustrato come ADO.NET fornisca un'ottima base per accedere ai dati. Oggetti come `DbConnection`, `DbCommand`, `DbDataReader` e `DataSet` offrono tutto ciò di cui abbiamo bisogno per interagire con un database.

Tuttavia, lavorare con questi oggetti in maniera diretta nel nostro codice significa legarlo al database e alla sua struttura. Per questo motivo, le applicazioni moderne sfruttano una tecnica più elaborata per manipolare i dati. I dati vengono recuperati dal database utilizzando le classi di ADO.NET (in uno strato dedicato all'accesso ai dati) e riversati in un insieme di classi (l'insieme è noto come object model, mentre le classi sono note come entity), che vengono restituite all'applicazione. L'applicazione manipola i dati contenuti nelle classi dell'object model e demanda allo strato di accesso ai dati la loro persistenza sul database. In questo modo, la nostra applicazione lavora principalmente con l'object model senza preoccuparsi della struttura del database se non in uno strato dedicato. Grazie all'astrazione creata dall'object model e dallo strato di accesso ai dati, la nostra applicazione è agnostica rispetto al database e quindi più semplice sia da sviluppare sia da manutenere.

Quando sviluppiamo applicazioni seguendo questo pattern, possiamo trovare un valido aiuto nei cosiddetti Object/Relational Mapper o O/RM (O/RM d'ora in poi). Un O/RM è un framework che permette di dialogare con il database astraendo le classi di ADO.NET e restituendo direttamente istanze di oggetti dell'object model. Permette anche di tracciare le modifiche fatte agli oggetti e di persisterle sul database.

In questo capitolo parleremo dell'O/RM prodotto da Microsoft: Entity Framework Core. Questo framework condivide molte cose con la sua controparte .NET (Entity Framework 6) ma, al momento della stesura di questo libro, non espone diverse funzionalità che lo rendono meno flessibile. Tuttavia, i miglioramenti a Entity Framework Core sono continui e ben presto sarà al livello di Entity Framework 6 e lo supererà. Ne sono un esempio i miglioramenti apportati a Entity Framework Core 2.1. Questa versione, infatti, colma molte delle lacune presenti nelle precedenti versioni e rende Entity Framework Core una valida scelta.

Cosa è un O/RM

Prima di iniziare la spiegazione di Entity Framework Core, è bene accennare cosa sia un O/RM e quale idea risieda alla base di questo strumento di sviluppo. Come detto poco sopra, mascherare la struttura e l'interazione con il database dietro alle classi, permette di costruire applicazioni fortemente disaccoppiate dal database; questa è un'ottima cosa a livello di semplicità di sviluppo e manutenzione. Attraverso l'uso di un O/RM possiamo creare delle classi che rappresentino il dominio della nostra applicazione, indipendentemente da come i dati sono strutturati nel database. Sarà poi compito di uno specifico strato dell'applicazione tradurre i risultati delle query in oggetti e, viceversa, tradurre gli oggetti in comandi per aggiornare il database..

Prendendo come esempio il database `Northwind`, possiamo creare le classi `Order`, `OrderDetail` e `Customer`. In questo caso, le classi hanno una struttura speculare con le tabelle del database, ma non sempre è così. La teoria che sta dietro agli oggetti è completamente diversa dalla teoria che è alla base dei dati relazionali e questa diversità porta spesso (ma non sempre) ad avere classi diverse dalle tabelle. Il primo esempio di questa diversità risiede nella diversa granularità. Un cliente, in genere, ha un indirizzo di fatturazione e uno di spedizione (che possono coincidere o meno). Per rappresentare questi dati nel database, creiamo una tabella `Customers` con i campi indirizzo, C.a.p., città e nazione ripetuti per entrambe le tipologie di indirizzo. Quando invece creiamo le classi, la cosa migliore è crearne una (`AddressInfo`) con le proprietà di un indirizzo e poi nella classe `Customer` aggiungere due proprietà (`BillingAddress` e `ShippingAddress`) di tipo `AddressInfo`. Questo significa avere una tabella lato database e due classi lato Object Model, come è mostrato nella Figura 10.1.

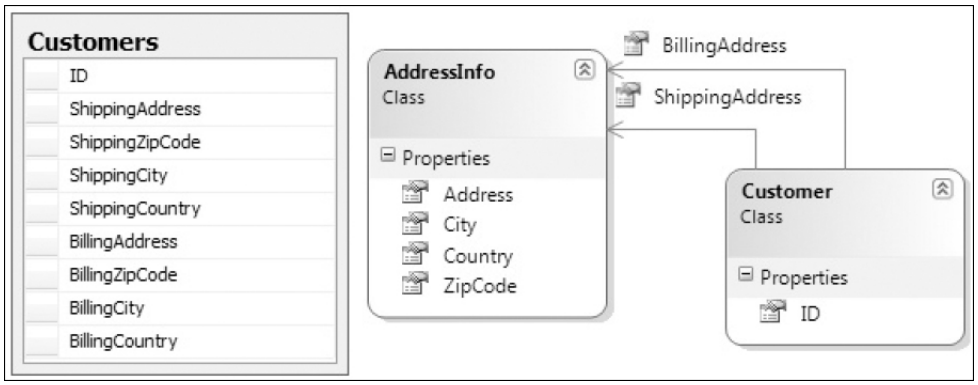


Figura 10.1 – La tabella clienti è descritta in due classi.

Un altro esempio è fornito dalla diversa modalità di relazione tra i dati. In un database, le relazioni tra i record sono mantenute tramite colonne con un vincolo di foreign key. Per esempio, per associare l'ordine a un cliente, mettiamo nella tabella degli ordini una colonna che

contenga l'id del cliente. Nell'object model le relazioni si esprimono usando direttamente gli oggetti. Quindi, per mantenere l'associazione tra l'ordine e il cliente, aggiungiamo la proprietà `Customer` alla classe `Order`, come è mostrato nella figura 10.2.

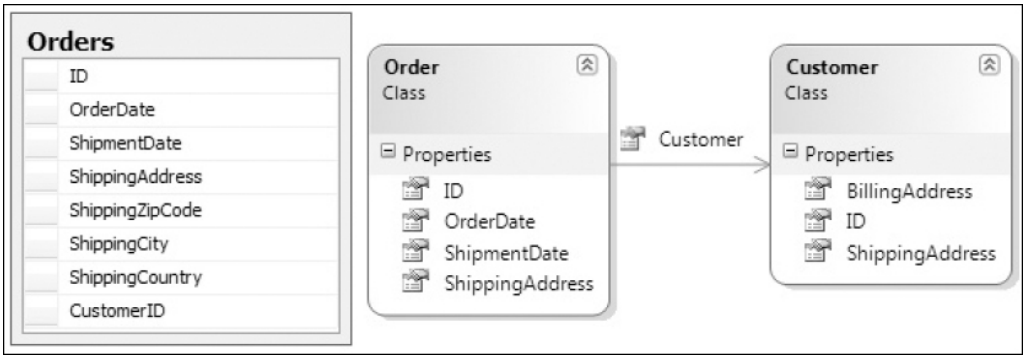


Figura 10.2 – La relazione tra ordini e clienti è mantenuta con una foreign key sul database e con una proprietà sul modello.

Le differenze si amplificano ulteriormente quando usiamo l'ereditarietà. Utilizzare questa tecnica nel mondo a oggetti è una cosa normalissima. Tuttavia, nel mondo relazionale non esiste il concetto di ereditarietà. Supponendo di avere un modello con le classi `Customer` e `Supplier` che ereditano dalla classe `Company`, come possiamo avere una simile rappresentazione nel database? Possiamo sicuramente creare degli artifici che ci permettano poi di ricostruire le classi, ma si tratta comunque di accorgimenti volti a coprire una diversità di fondo tra il mondo relazionale e quello a oggetti.

Risolvere manualmente tutte queste complessità (e anche altre) non è affatto banale ed è per questo motivo che, da tempo, esistono dei framework che coprono queste e altre necessità. Questi framework prendono il nome di O/RM, in quanto lavorano come (M)apper tra (O)ggetti e dati (R)elazionali. In questo modo le diversità tra i due mondi sono gestite dall'O/RM, lasciandoci liberi di preoccuparci del solo codice di business.

Gli O/RM agiscono come mapper, cioè mappano le classi e le relative proprietà con le tabelle e le colonne nel database. Il vantaggio che ne deriva è nel fatto che possiamo evitare di scrivere query verso il database, ma possiamo scriverle verso l'Object Model in un linguaggio specifico dell'O/RM, che poi si preoccuperà di creare il codice SQL necessario. In termini di logica di business questo rappresenta un enorme vantaggio, in quanto gli oggetti rappresentano la logica in maniera molto più semplice delle tabelle. Lo stesso ragionamento vale per gli aggiornamenti sul database. Noi ci preoccupiamo solo di modificare gli oggetti e poi demandiamo all'O/RM la persistenza di questi sul database. Ora dovrebbe essere più chiaro cosa significhi avere un'applicazione disaccoppiata dal database.

In conclusione, un O/RM è una parte di software molto potente ma, allo stesso tempo, molto complessa e pericolosa, poiché il livello di astrazione che introduce rischia di farci dimenticare che c'è un database, e questo è negativo. Dobbiamo sempre controllare le query generate dall'O/RM e verificare le istruzioni di manipolazione dati, per essere sicuri che le performance corrispondano ai requisiti.

Ora che abbiamo capito quali compiti svolge un O/RM possiamo vedere come questi compiti siano svolti da Entity Framework Core e come possiamo usare questo strumento per semplificare lo sviluppo del codice di accesso ai dati. Il primo passo consiste nel creare le classi del modello a oggetti per poi mapparle sul database.

Mappare il modello a oggetti sul database

Visto che la M di O/RM sta per Mapper, è facile immaginare che la fase di mapping tra il modello a oggetti e il database sia molto importante. Per mappare le classi verso il database, dobbiamo svolgere tre attività: prima scriviamo le classi, poi creiamo la classe di contesto (semplicemente contesto d'ora in poi) e quindi, tramite quest'ultima, mappiamo le classi.

L'operazione di mapping viene effettuata tramite codice, utilizzando o specifiche API o decorando con attributi le classi e le proprietà dell'object model. Inoltre, se scriviamo i nomi delle classi, delle loro proprietà e delle proprietà del contesto sfruttando determinate convenzioni, non abbiamo nemmeno la necessità di scrivere il codice di mapping, in quanto Entity Framework Core è già in grado di dedurre automaticamente dai nomi come il modello debba essere mappato. Le API permettono di effettuare qualunque tipologia di mapping che Entity Framework Core metta a disposizione, mentre gli attributi e le convenzioni permettono di mappare solo un sottoinsieme delle possibilità di Entity Framework Core. Per questo motivo parleremo in maniera approfondita solo del mapping tramite API.

Vediamo ora come creare e mappare le classi introdotte nelle precedenti sezioni.

Disegnare le classi

Scrivere una classe dell'object model è estremamente semplice in quanto consiste nel creare una classe con delle proprietà, esattamente come faremmo per qualunque altra classe. Non è richiesta alcuna integrazione con Entity Framework Core, come possiamo notare nel codice dell'Esempio 10.1.

Esempio 10.1

```
public class AddressInfo
{
   public string Address { get; set; }
   public string City { get; set; }
   public string Region { get; set; }
   public string PostalCode { get; set; }
   public string Country { get; set; }
}

public partial class Customer
{
   public Customer()
   {
      Orders = new HashSet<Order>();
   }
   public string CustomerId { get; set; }
   public string CompanyName { get; set; }
   public string ContactName { get; set; }
   public string ContactTitle { get; set; }
   public AddressInfo Address { get; set; }
   public string Phone { get; set; }
   public string Fax { get; set; }
   public ICollection<Order> Orders { get; set; }
}

public class Order
{
   public Order()
   {
      OrderDetails = new HashSet<OrderDetail>();
   }
   public int OrderId { get; set; }
   public string CustomerId { get; set; }
   public int? EmployeeId { get; set; }
   public DateTime? OrderDate { get; set; }
   public DateTime? RequiredDate { get; set; }
   public DateTime? ShippedDate { get; set; }
   public int? ShipVia { get; set; }
   public decimal? Freight { get; set; }
   public string ShipName { get; set; }
   public AddressInfo ShipAddress { get; set; }
   public Customer Customer { get; set; }
   public ICollection<OrderDetail> OrderDetails { get; set; }
}

public partial class OrderDetail
{
   public int OrderId { get; set; }
   public int ProductId { get; set; }
   public decimal UnitPrice { get; set; }
   public short Quantity { get; set; }
   public float Discount { get; set; }
   public Order Order { get; set; }
}
```

Il codice mostra chiaramente alcune caratteristiche del nostro modello:

- le classi sono semplici classi POCO (Plain Old CLR Object), con proprietà che rappresentano i dati e nessuna relazione con l'O/RM. Questo modello potrebbe essere usato da Entity Framework Core così come da altri O/RM senza bisogno di alcuna modifica;

- le relazioni sono espresse tramite proprietà (dette Navigation Property) che si riferiscono direttamente a un oggetto in caso di relazioni uno a uno (come da dettaglio a ordine), o una lista di oggetti in casi di relazioni uno a molti (come da ordine a dettagli);

- il tipo `AddressInfo` è un tipo senza una chiave, referenziato da altri oggetti. Questo tipo di classe è detta owned type e agisce come semplice contenitore di proprietà e non come tipo da mappare verso una tabella;

- anche se non presenti nell'esempio, gli `enum` sono supportati come qualunque altro tipo nativo.

Ora che abbiamo visto il codice dell'object model andiamo a vedere il codice del contesto.

Creare il contesto

Il contesto è la classe che agisce da ponte tra il mondo a oggetti dell'object model e il mondo relazionale del database. Infatti, è attraverso questa classe che possiamo mappare l'object model verso il database ed effettuare tutte le operazioni, siano esse query o modifiche di dati negli oggetti.

Il contesto è una classe che eredita da `DbContext` e che definisce una proprietà di tipo `DbSet<T>`, chiamata entityset, per ogni classe dell'object model mappata verso una tabella del database (il tipo `T` corrisponde al tipo della classe). Nel nostro caso, il contesto conterrà tre proprietà: una per la classe `Customer`, una per la classe `Order` e una per la classe `Order_Detail`. Queste proprietà rappresentano il punto di entrata per recuperare e modificare oggetti nel database.

> *Nel caso in cui usiamo l'ereditarietà nel nostro object model (per esempio una classe Company da cui derivano Customer e Supplier), abbiamo un solo entityset per tutta la gerarchia. Il tipo dell'entityset è quello della classe base.*

Oltre a queste proprietà, il contesto definisce anche un costruttore che accetta in input un oggetto `DbContextOptions<T>`, dove T è il tipo del contesto, che viene passato in input al costruttore base. Questo costruttore è fondamentale per l'integrazione con ASP.NET come vedremo in seguito. L'Esempio 10.2 mostra il codice del contesto.

Esempio 10.2

```
public class NorthwindContext : DbContext
{
   public NorthwindContext(DbContextOptions<NorthwindContext> o) : base(o) { }
   public DbSet<Customer> Customers { get; set; }
   public DbSet<OrderDetail> OrderDetails { get; set; }
   public DbSet<Order> Orders { get; set; }
   protected override void OnConfiguring(DbContextOptionsBuilder o)
   {
   }
}
```

Possiamo configurare la classe di contesto eseguendo l'override del metodo `OnConfiguring`, che accetta un oggetto di tipo `DbContextOptionsBuilder`, che espone i parametri di configurazione. Ora che abbiamo visto come creare la classe di contesto, vediamo come eseguire il mapping delle classi verso il database attraverso questa classe.

Mapping tramite convenzioni

Quando un contesto viene istanziato la prima volta e viene eseguita la prima operazione, Entity Framework Core esegue il codice di mapping. Per prima cosa vengono analizzate le proprietà di tipo `DbSet<T>` del contesto per recuperarne le relative classi e verificare se queste rispettino determinate convenzioni che ne permettono il mapping senza necessità di scrivere codice. Per chiarire meglio questo concetto, analizziamo le convenzioni applicate alla classe `Order`:

- La classe viene mappata verso una tabella che ha il nome dell'entityset. Nel nostro caso, la classe viene mappata verso la tabella `Orders`;

- le colonne della tabella hanno lo stesso nome delle proprietà della classe. Nel caso di proprietà all'interno di owned type, il nome del campo sulla tabella viene calcolato unendo il nome della proprietà di tipo owned type con quello della proprietà al suo interno e separandoli con il carattere "_". Nel nostro caso, la proprietà `City` all'interno della proprietà `ShipAddress` viene mappata sulla colonna `ShipAddress_City`. Se un owned type contiene un altro owned type, la tecnica di calcolo non cambia e i nomi vengono concatenati;

- se una proprietà si chiama, indipendentemente dal case, `ID` o `{NomeClasse}ID` (dove il segnaposto NomeClasse viene rimpiazzato dal nome della classe che contiene la proprietà) questa viene automaticamente eletta a chiave primaria della classe. Se la proprietà è di tipo intero, questa è trattata come Identity. Nel nostro caso, la proprietà `OrderId` è automaticamente identificata come chiave primaria;

- Il tipo del campo su cui la proprietà è mappata è analogo al tipo della proprietà (int per i tipi `Int32`, bit per i tipi `Boolean` e così via). Le proprietà di tipo `Nullable<T>` e le proprietà di tipo `String` sono considerate `null` sul database, le altre proprietà sono considerate `not null`. Infine, le proprietà di tipo `String` sono considerate unicode a lunghezza massima. Nel nostro caso, la proprietà `CustomerId` è mappata su una colonna di tipo `int`, mentre la proprietà `ShipName` è mappata su una colonna di tipo `nvarchar`;

- Le navigation property con una reference uno a uno vengono automaticamente mappate utilizzando come nome della colonna il nome della proprietà chiave della classe a cui la proprietà si riferisce. Nel nostro caso, la navigation property `Customer` punta a un oggetto di tipo `Customer` la cui proprietà chiave è `CustomerId.` Questo significa che Entity Framework Core sfrutta la colonna `CustomerId` nella tabella `Orders` per mappare la relazione tra le due entità.

Alla luce di queste convenzioni appare chiaro che buona parte del codice di mapping può essere gestito automaticamente dalle convenzioni. Tuttavia c'è una parte di mapping che va gestita a mano come la lunghezza massima delle stringhe, la configurazione dei nomi dei campi quando sono diversi dalle proprietà, la configurazione di chiavi primarie composte da più proprietà, gli indici, le foreign key e altro ancora.

Mapping tramite API

Per mappare una classe verso il database usando le API di Entity Framework Core, dobbiamo eseguire l'override del metodo `OnModelCreating` nella classe di contesto. Questo metodo accetta in input un oggetto di tipo `ModelBuilder`. Il metodo principale di questa classe è `Entity`, che accetta il tipo della classe come parametro generico e un metodo che specifica il mapping della classe. Questo metodo prende in input un oggetto di tipo `EntityTypeBuilder<T>,` dove `T` rappresenta la classe, tramite cui possiamo eseguire tutte le operazioni di mapping della classe. I metodi principali della classe `EntityTypeBuilder<T>` sono esposti nella seguente tabella.

Tabella 10.1 – Metodi di mapping di EntityTypeBuilder<T>.

Metodo	Scopo
`Property`	Accetta una lambda che rappresenta una proprietà della classe e restituisce un oggetto che la rappresenta e sul quale possiamo eseguire metodi per mapparne le caratteristiche (nome della colonna sulla tabella, dimensioni, precisione, nullabilità e così via)
`HasIndex`	Accetta una lambda che rappresenta una o più proprietà che formano un indice sulla tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche dell'indice (nome, univocità)
`ToTable`	Specifica il nome della tabella su cui la classe viene mappata nel database
`OwnsOne`	Accetta una lambda che rappresenta una proprietà della classe che si riferisce a un owned type e restituisce un oggetto che la rappresenta. Su questo oggetto possiamo usare metodi per mappare le proprietà dell'owned type. Questi metodi sono quasi gli stessi visti in questa tabella.
`HasOne/HasMany`	Accettano una lambda che rappresenta una navigation property della classe e restituisce un oggetto che permette di specificarne le caratteristiche di mapping (nome della colonna sul db, relazione con la classe corrispondente, nome della foreign key).
`HasFilter`	Accetta una lambda che specifica un filtro che verrà applicato a ogni query che riguarda la tabella, senza bisogno di specificarlo in ogni query
`HasKey`	Accetta una lambda che rappresenta una o più proprietà che formano la chiave primaria della tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche della chiave (nome, clustered se il database è SqlServer)

I metodi di `EntityTypeBuilder<T>` si dividono in due categorie: quelli che mappano informazioni tra la classe e la tabella (`HasKey`, `ToTable`, `HasFilter`) e quelle che tornano le proprietà per poi specificarne il mapping (`Property`, `OwnsOne`, `HasOne`, `HasMany`). Vediamo nella prossima tabella quali sono i metodi di mapping relativi alle proprietà.

Tabella 10.2 – Metodi di mapping.

Metodo	Applicabile su	Scopo
`HasMaxLength`	`String`	Specifica la lunghezza massima del campo
`IsUnicode`	`String`	Specifica se il campo supporta caratteri unicode
`HasColumnName`	Tutti i tipi	Specifica il nome del campo mappato

HasColumnType	Tutti i tipi	Specifica il tipo del campo mappato
IsRequired	Tutti i tipi	Specifica se il campo è null o no (not null per default)
IsConcurrencyToken	Tutti i tipi	Specifica che la proprietà fa parte del token per gestire la concorrenza ottimistica negli aggiornamenti
IsRowVersion	Tutti i tipi	Specifica che la proprietà contiene la versione della riga (ogni database può interpretare il campo mappato in modo diverso)
ValueGeneratedNever	Tutti i tipi	Specifica che il valore della proprietà viene generato dal client (la nostra applicazione)
ValueGeneratedOnAdd	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client in fase di inserimento, ma spetta al database decidere se usare quello del client o generarne un altro. Per fare un esempio, se usiamo un'identity con SqlServer, un eventuale valore fornito dal client viene scartato e sostituito con quello generato dal server.
ValueGeneratedOnAddOrUpdate	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client sia in fase di inserimento sia in fase di aggiornamento. Così come per ValueGeneratedOnAdd, spetta al database decidere se usare il valore inviato dal del client o generarne un altro.
HasDefaultValueSql	Tutti i tipi	Specifica il valore di default della proprietà sul database.

I metodi di mapping appena elencati sono agnostici rispetto al tipo di database. Il provider di Entity Framework Core per SqlServer aggiunge ulteriori metodi specifici per il database (identity, sequence e altro). Altri provider per altri database possono aggiungere altri metodi (tramite extension method) per consentire altre modalità di mapping specifiche per le loro esigenze.

Ora che abbiamo illustrato i metodi di mapping, vediamo come applicarli per mappare la classe AddressInfo all'interno del metodo OnModelCreating.

Esempio 10.3

```
modelBuilder.Entity<AddressInfo>(entity =>
{
   entity.Property(e => e.City).HasMaxLength(15);
   entity.Property(e => e.Country).HasMaxLength(15);
   entity.Property(e => e.PostalCode).HasMaxLength(10);
   entity.Property(e => e.Region).HasMaxLength(15);
   entity.Property(e => e.Address).HasMaxLength(60);
});
```

In questo esempio sfruttiamo il metodo Property per recuperare il riferimento a una proprietà dell'owned type e specifichiamo la massima lunghezza sfruttando il metodo HasMaxLength. Il tipo AddressInfo è piuttosto semplice da mappare, quindi possiamo passare al prossimo: Customer.

Esempio 10.4

```
modelBuilder.Entity<Customer>(entity =>
{
   entity.HasKey(e => e.CustomerId);
   entity.Property(e => e.CustomerId)
      .HasColumnType("nchar(5)")
      .ValueGeneratedNever();
   entity.HasIndex(e => e.CompanyName)
      .HasName("CompanyName");
   entity.OwnsOne(e => e.Address)
      .Property(e => e.Address)
      .HasColumnName("Address");
   entity.OwnsOne(e => e.Address)
      .Property(e => e.City)
      .HasColumnName("City");
   entity.OwnsOne(e => e.Address)
      .Property(e => e.Country)
      .HasColumnName("Country");
   entity.OwnsOne(e => e.Address)
```

```
      .Property(e => e.PostalCode)
      .HasColumnName("PostalCode");
   entity.OwnsOne(e => e.Address)
      .Property(e => e.Region)
      .HasColumnName("Region");
   entity.OwnsOne(e => e.Address)
      .HasIndex(e => e.City)
      .HasName("City");
   entity.OwnsOne(e => e.Address)
      .HasIndex(e => e.PostalCode)
      .HasName("PostalCode");
   entity.Property(e => e.CompanyName)
      .IsRequired()
      .HasMaxLength(40);
   entity.Property(e => e.ContactName).HasMaxLength(30);
   entity.Property(e => e.ContactTitle).HasMaxLength(30);
   entity.Property(e => e.Fax).HasMaxLength(24);
   entity.Property(e => e.Phone).HasMaxLength(24);
});
```

La classe `Customer` è decisamente più complessa rispetto a `AddressInfo`. Innanzitutto specifichiamo che la proprietà che mappa sulla chiave primaria è `CustomerId`. Questo mapping non sarebbe necessario in quanto Entity Framework Core è in grado di capirlo dalle convenzioni, ma è comunque riportato per dare un esempio del suo utilizzo.

Successivamente alla specifica della chiave primaria, viene specificato il mapping della proprietà che agisce come chiave primaria sfruttando il metodo `HasColumnType`, per indicare il tipo del campo sulla tabella, e `ValueGeneratedNever` per indicare che il valore viene sempre generato dal client. Nell'istruzione che segue, tramite il metodo `HasIndex`, specifichiamo che la tabella ha un indice, che la colonna `CompanyName` fa parte dell'indice e che l'indice si chiama come la colonna (metodo `HasName`). Possiamo specificare che un indice è univoco aggiungendo la chiamata al metodo `IsUnique`.

Le istruzioni successive mappano le proprietà dell'owned type `AddressInfo` verso la tabella `Customers`. Qui è interessante il modo in cui recuperiamo le proprietà. Questo recupero potrebbe avvenire con una unica lambda, ma, poiché Entity Framework Core non la supporta, dobbiamo prima recuperare la proprietà `Address` all'interno di `Customer` e poi recuperare le sue proprietà interne, per mapparle verso la relativa colonna con il metodo `HasColumnName`. Quest'operazione è necessaria in quanto in sua assenza Entity Framework Core utilizzerebbe le convenzioni e userebbe i nomi `Address_City`, `Address_Address` e così via, come nomi delle colonne sul database, generando quindi un'eccezione a run time, in quanto queste colonne non esistono.

Oltre a specificare i nomi delle colonne che mappano sulle proprietà dell'owned type, specifichiamo anche gli indici presenti su tali colonne. Infine, usiamo i metodi `IsRequired` e di nuovo `HasMaxLength` per specificare le caratteristiche delle proprietà rimaste.

Ora cambiamo classe e analizziamo il mapping di `Order` che viene mostrato nell'Esempio 10.5.

Esempio 10.5

```
modelBuilder.Entity<Order>(entity =>
{
   entity.HasKey(e => e.OrderId);
   entity.HasIndex(e => e.CustomerId).HasName("CustomersOrders");
   entity.HasIndex(e => e.EmployeeId).HasName("EmployeesOrders");
   entity.HasIndex(e => e.OrderDate).HasName("OrderDate");
   entity.OwnsOne(e => e.ShipAddress)
      .HasIndex(e => e.PostalCode)
      .HasName("ShipPostalCode");
   entity.OwnsOne(e => e.ShipAddress)
      .Property(e => e.Address)
      .HasColumnName("ShipAddress");
   entity.OwnsOne(e => e.ShipAddress)
      .Property(e => e.City)
      .HasColumnName("ShipCity");
   entity.OwnsOne(e => e.ShipAddress)
      .Property(e => e.Country)
      .HasColumnName("ShipCountry");
   entity.OwnsOne(e => e.ShipAddress)
      .Property(e => e.PostalCode)
      .HasColumnName("ShipPostalCode");
   entity.OwnsOne(e => e.ShipAddress)
      .Property(e => e.Region)
      .HasColumnName("ShipRegion");
```

```
    entity.HasIndex(e => e.ShipVia)
       .HasName("ShippersOrders");
    entity.HasIndex(e => e.ShippedDate)
       .HasName("ShippedDate");
    entity.Property(e => e.OrderId).HasColumnName("OrderID");
    entity.Property(e => e.EmployeeId).HasColumnName("EmployeeID");
    entity.Property(e => e.Freight)
       .HasColumnType("money")
       .HasDefaultValueSql("((0))");
    entity.Property(e => e.OrderDate).HasColumnType("datetime");
    entity.Property(e => e.RequiredDate).HasColumnType("datetime");
    entity.Property(e => e.ShipName).HasMaxLength(40);
    entity.Property(e => e.ShippedDate).HasColumnType("datetime");
    entity.HasOne(d => d.Customer).WithMany(p => p.Orders);
 });
```

Il mapping della classe `Order` utilizza molti dei metodi già visti nei precedenti esempi. Infatti, vengono prima specificati gli indici, quindi mappate le proprietà dell'indirizzo e poi mappate le proprietà rimanenti. In particolare, per la proprietà `Freight` viene usato il metodo `HasDefaultValueSql`, il quale specifica il valore di default del campo sulla tabella del database.

Un altro metodo interessante è `HasOne`. Questo viene utilizzato per mappare la navigation property `Customer` verso l'omonima classe. Per convenzione, Entity Framework Core usa `CustomerId` come nome della colonna che agisce da foreign key verso la tabella `Customers`. Questo perché la colonna che rappresenta la chiave primaria sulla tabella `Customers` si chiama `CustomerId`. Possiamo modificare questo comportamento usando il metodo `HasForeignKey` e passando in input il nome della colonna sulla tabella `Orders`. Il successivo metodo `WithMany` specifica che la navigation property `Orders` di `Customer` è mappata verso `Order` sfruttando la stessa foreign key.

Ora non rimane che mappare la classe `OrderDetail`.

Esempio 10.6

```
 modelBuilder.Entity<OrderDetail>(entity =>
 {
    entity.ToTable("Order Details");
    entity.HasKey(e => new { e.OrderId, e.ProductId });
    entity.HasIndex(e => e.OrderId).HasName("OrdersOrder_Details");
    entity.HasIndex(e => e.ProductId).HasName("ProductsOrder_Details");
    entity.Property(e => e.OrderId).HasColumnName("OrderID");
    entity.Property(e => e.ProductId).HasColumnName("ProductID");
    entity.Property(e => e.Discount).HasDefaultValueSql("((0))");
    entity.Property(e => e.Quantity).HasDefaultValueSql("((1))");
    entity.Property(e => e.UnitPrice)
       .HasColumnType("money")
       .HasDefaultValueSql("((0))");
    entity.HasOne(d => d.Order)
       .WithMany(p => p.OrderDetails)
       .OnDelete(DeleteBehavior.ClientSetNull)
 });
```

Il mapping di `OrderDetail` utilizza altri metodi molto importanti.

Il primo è `ToTable` che specifica il nome della tabella verso cui la classe mappa. In questo caso, siamo obbligati a usare questo metodo in quanto il nome della tabella dedotto dalle convenzioni, `OrderDetails`, è errato.

Il secondo è `HasKey`. Sebbene abbiamo già visto questo metodo prima, in questo caso il suo utilizzo mostra come specificare una chiave composta da più proprietà. Infatti, il metodo non ritorna una sola proprietà, bensì un anonymous type con le proprietà che fanno parte della chiave primaria. Questa stessa tecnica è utilizzabile anche nel metodo `HasIndex` per specificare indici composti.

Come abbiamo detto in precedenza, oltre che tramite convenzioni e API, esiste un terzo metodo di mapping che utilizza le data annotation e di cui ci occuperemo nella prossima sezione.

Mapping tramite data annotation

Il mapping tramite le data annotation prevede che in testa alle classi dell'object model e alle proprietà siano presenti alcuni attributi che vengono interpretati dal motore di Entity Framework Core. Questi attributi permettono di specificare il nome della tabella su cui una classe mappa, il nome della colonna su cui una proprietà mappa, il tipo specifico della colonna, se è una primary key, se è obbligatoria, la sua lunghezza massima e altro ancora. Sebbene sia comodo, il mapping tramite data annotation non copre tutte le esigenze, come invece fa il mapping tramite API. La Tabella 10.3 mostra le principali data annotation.

Tabella 10.3 – Data annotation per il mapping.

Attributo	Applicabile su	Scopo
`Table`	Classe	Specifica la tabella su cui la classe mappa
`Key`	Proprietà	Specifica che la proprietà fa parte della chiave primaria
`Column`	Proprietà	Specifica il nome e il tipo della colonna su cui la proprietà mappa
`DatabaseGenerated`	Proprietà	Specifica se la colonna è su cui la proprietà mappa è un'identity, calcolata o normale
`Required`	Proprietà	Specifica che la proprietà è obbligatoria
`StringLength`	Proprietà	Specifica la lunghezza massima della proprietà

A prescindere dalla tecnica di mapping utilizzata, abbiamo scritto il codice necessario per cominciare a utilizzare Entity Framework Core. Tuttavia, finora siamo ricorsi alla scrittura manuale del codice, ma Entity Framework Core permette di generare il codice partendo dalle tabelle del database.

Generare automaticamente le classi dal database

Quando abbiamo già il database a disposizione e dobbiamo generare l'object model, possiamo utilizzare il comando `scaffold-dbcontext`. Questo comando prende in input la stringa di connessione al database, il provider di Entity Framework da usare e genera una classe per ogni tabella, mappando automaticamente le proprietà e le navigation property, e generando anche la classe di contesto con tutti gli entityset.

> *Oltre ai parametri base, possiamo specificare anche di quali tabelle vogliamo eseguire il mapping, in quale cartella mettere i file generati, il nome del contesto, se generare un log di generazione e altro ancora. Per conoscere tutti i parametri, rimandiamo alla pagina della documentazione, disponibile all'url:* `http://aspit.co/bk0`.

Nell'Esempio 10.7 possiamo vedere il comando da utilizzare per generare le classi, il contesto e il mapping dell'object model che abbiamo analizzato.

Esempio 10.7

```
scaffold-dbcontext
   -connection "connection string"
   -provider "Microsoft.EntityFrameworkCore.SqlServer"
   -tables "Customers", "Order Details", "Orders"
   -output "data"
   -context "NorthwindContext"
 -verbose
```

Questo comando va lanciato dalla finestra Package Manager Console di Visual Studio, all'interno della quale va prima selezionato su quale progetto creare i file e poi lanciato il comando.

Se non usiamo Visual Studio, possiamo impiegare le estensioni di Entity Framework Core per la CLI di .NET: `dotnet ef`. Queste estensioni sono installate globalmente, quindi non necessitano di pacchetti NuGet da installare.

Esempio 10.8

```
dotnet ef dbcontext scaffold
   "connection string"
   Microsoft.EntityFramework.SqlServer
   -t "Customers", "Order Details", "Orders"
   -o "data"
   -c "NorthwindContext"
 -v
```

Il generatore crea classi e proprietà mappandole uno a uno con le tabelle e le colonne del database. Questo significa che non prende in considerazione la creazione di owned type come `AddressInfo`.

> *Oltre al comando per generare le classi e il contesto partendo dal database, ne esistono molti altri. Esistono comandi per manipolare e applicare le migrazioni, per eliminare il database e per ottenere informazioni sui contesti presenti nel progetto.*

Avere a disposizione il codice di mapping (sia esso scritto a mano o generato da Entity Framework Core) è la prima parte di ciò che serve per far funzionare Entity Framework Core; la seconda è la configurazione del contesto in ASP.NET.

Configurare Entity Framework Core con ASP.NET

Come abbiamo visto nei precedenti capitoli, ASP.NET è fortemente basato sulla dependency injection. Poiché il contesto è una dipendenza dei controller, possiamo aggiungerlo come parametro del costruttore del controller e lasciare che sia il motore di dependency injection di ASP.NET a gestirne il ciclo di vita. La configurazione del contesto avviene nel metodo `ConfigureServices` della classe `Startup`, all'interno del quale usiamo il metodo `AddDbContext`. Questo metodo accetta in input una funzione, la quale accetta un parametro che rappresenta le opzioni del contesto e configura tali opzioni. Quelle principali sono il tipo di database e la stringa di connessione, senza i quali non sarebbe possibile utilizzare il contesto. Queste opzioni vengono poi passate al costruttore del contesto. L'Esempio 10.9 mostra il codice necessario alla configurazione.

Esempio 10.9

```
services.AddDbContext<NorthwindContext>(o =>
{
   var connectionString = "stringa di connessione da configurazione";
   o.UseSqlServer(connectionString);
});
```

Il metodo `UseSqlServer` specifica che intendiamo usare il provider di Entity Framework Core per SqlServer e la stringa in input rappresenta la stringa di connessione da usare. Per default, il metodo registra il contesto come `Scoped` così che esista una sola istanza per richiesta web. Volendo possiamo modificare questo comportamento passando un secondo parametro al metodo `AddDbContext` di tipo `ServiceLifetime` e specificando un lifetime diverso.

> *Avere un'istanza del contesto per richiesta è il pattern più comune. A seconda delle esigenze, si può preferire gestire manualmente il ciclo di vita del contesto per avere più istanze in una richiesta, ma non si deve mai avere un contesto singletone perchè questo verrebbe condiviso da tutti gli utenti dell'applicazione, generando comportamenti imprevedibili.*

Al posto del metodo `AddDbContext` possiamo usare anche `AddDbContextPool`. Questo metodo permette una piccola ottimizzazione di performance. Con `AddDbContext`, ASP.NET crea ogni volta un'istanza nuova. Sebbene l'istanziazione di un contesto sia abbastanza leggera, ha comunque un costo. Il metodo `AddDbContextPool` genera un pool di contesti e ogni volta che ASP.NET deve generarne uno, invece che crearlo da zero lo va a prendere dal pool. In questo modo abbiamo una maggior lentezza in fase di startup ma una maggiore velocità durante l'esecuzione delle richieste. L'unica accortezza da mantenere è quella di non salvare alcuno stato nel contesto, altrimenti quello stato verrà usato anche dalle richieste che successivamente sfrutteranno quella specifica istanza del contesto. La firma di `AddDbContextPool` è la stessa di `AddDbContext`, quindi il suo utilizzo è estremamente semplice.

A questo punto siamo pronti per utilizzare Entity Framework Core.

Recuperare i dati dal database

La ricerca di dati nel database avviene tramite gli entityset del contesto, quindi la prima cosa da fare è recuperare l'istanza del contesto. Grazie alla sua configurazione nel sistema di dependency injection di ASP.NET, possiamo ottenerlo in input nel costruttore oppure come parametro della action, aggiungendo al parametro l'attributo `FromServices`.

Esempio 10.10

```
public class HomeController
{
   NorthwindContext _ctx;
   public HomeController(NorthwindContext ctx) {
      _ctx = ctx;
   }
}

public class HomeController
{
   public IActionResult Index([FromServices] NorthwindContext ctx)
   {
   }
}
```

Ora che abbiamo il contesto, dobbiamo sfruttare i suoi entityset. Poiché il tipo `DbSet<T>` implementa indirettamente l'interfaccia `IQueryable<T>`, questo espone tutti gli extension method di LINQ. Questo non significa assolutamente che i dati siano in memoria. Il provider LINQ di Entity Framework Core intercetta l'esecuzione della query verso l'entityset e la trasforma in un *expression tree,* che viene poi convertito in SQL grazie alle informazioni di mapping. Il risultato è che noi scriviamo query LINQ e tutto il lavoro di conversione è affidato a Entity Framework Core.

L'Esempio 10.11 mostra come sia semplice recuperare la lista dei clienti italiani.

Esempio 10.11

```
var customers1 = from c in ctx.Customers
                     where c.Address.Country == "Italy"
                     select c;
var customers2 = ctx.Customers.Where(c => c.Address.Country == "Italy");
```

L'Esempio 10.11 porta ad alcune importanti considerazioni. La prima è che non dobbiamo gestire né connessioni né transazioni né altri oggetti legati all'interazione col database. Entity Framework Core astrae tutto per noi, restituendoci direttamente oggetti.

La seconda è che il codice che abbiamo appena visto non esegue alcuna query. La deferred execution è perfettamente supportata dal provider LINQ di Entity Framework Core, quindi, finché non enumeriamo fisicamente i risultati, la query non viene effettuata sul database. È importante sottolineare che a causa di questo comportamento, se eseguiamo la query quando il contesto è stato eliminato, otteniamo un'eccezione a run time. Il caso più comune in cui ci troviamo in questa situazione è quando assegniamo la variabile `customer1` a una proprietà del modello e poi nella view enumeriamo la proprietà. La query viene scatenata quando enumeriamo la proprietà ma, quando siamo nel contesto di esecuzione della view, il nostro contesto è già stato eliminato, in quanto siamo usciti dal suo scope.

La terza cosa da notare è che nelle query possiamo utilizzare sia la query syntax sia la normale sintassi basata sugli extension method, ma nel secondo caso dobbiamo prestare attenzione a quali extension method utilizzare. Il provider di Entity Framework Core non supporta tutti i metodi LINQ e i relativi overload poiché alcuni non trovano una controparte sui database, mentre altri non hanno la possibilità di essere tradotti in SQL. Gli extension method disponibili sono quelli di aggregazione, di intersezione, di ordinamento, di partizionamento, di proiezione, di raggruppamento oltre a quelli di insieme.

Per esempio, quando vogliamo recuperare un solo oggetto, i metodi `First` e `Single` sono validi. La differenza risiede nel fatto che mentre `First` viene tradotto in una `TOP 1`, `Single` viene tradotto in una `TOP 2`, per verificare che non sia presente più di un elemento. Inoltre se la ricerca dell'oggetto avviene per chiave primaria, possiamo anche utilizzare il metodo `Find` (o la sua controparte asincrona `FindAsync`) della classe `DbSet<T>` come mostrato nel prossimo esempio.

Esempio 10.12

```
var c1 = ctx.Customers.First(c => c.CustomerID == "ALFKI");
var c2 = ctx.Customers.Find("ALFKI");
var c3 = await ctx.Customers.FindAsync("ALFKI");
```

Una cosa che torna utilissima in LINQ sono le proiezioni. Molto spesso non abbiamo bisogno di tutta la classe, ma solo di alcuni suoi dati. Specificando in una proiezione quali proprietà dobbiamo estrarre (come nell'Esempio 10.13), faremo in modo che l'SQL generato estragga solo quelle, ottenendo così un'ottimizzazione delle performance.

Esempio 10.13

```
ctx.Customers.Select(c => new { c.CustomerID, c.CompanyName});
```

Esempio 10.13 – SQL

```
SELECT
   c.CustomerID,
   c.CompanyName
FROM [dbo].[Customers] AS c
```

Il provider LINQ di Entity Framework Core è un argomento controverso. Da un lato, le potenzialità del provider sono enormi, in quanto traduce molti metodi LINQ in SQL, supporta l'esecuzione mista di codice LINQ che genera SQL e codice LINQ eseguito in locale, permette di mischiare parzialmente codice SQL e codice LINQ per generare una unica query SQL sul server, e altro ancora. Dall'altro lato, alcuni metodi di LINQ non sono supportati, o sono solo parzialmente supportati, mentre altri si comportano in modo inaspettato, con conseguenze non facili da identificare specie se si è alle prime armi. Fortunatamente, Entity Framework Core permette di lanciare un'eccezione ogni volta che esegue una query che scarica i dati in locale e li processa invece di generare codice SQL. L'Esempio 10.14 mostra come fare.

Esempio 10.14

```
protected override void OnConfiguring(DbContextOptionsBuilder ob)
{
   ob.ConfigureWarnings(warnings =>
      warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
}
```

Il metodo `ConfigureWarnings` accetta un delegato che viene invocato ogni volta che il motore solleva un warning. Nel delegato specifichiamo che quando il tipo di warning è relativo a una valutazione della query sul client (ovvero i dati vengono scaricati sul client e poi processati) deve essere sollevata un'eccezione. In questo modo possiamo evitare possibili problemi di performance.

In altri casi, le query possono soffrire del problema 1+n, rallentando vistosamente le performance dell'applicazione. Un caso in cui questo accade è quando usiamo una projection che include campi di una navigation property che punta a una lista di oggetti.

Prendiamo in esame il seguente codice.

Esempio 10.15

```
var orders = ctx.Orders
   .Where(o => o.Customer.CustomerId == ”ALFKI”)
   .Select(c => new {
      CustomerId = c.Customer.CustomerId,
      Products = c.OrderDetails.Select(d => d.ProductId)
   })
   .ToList (); // query che estrae dati ordine
foreach (var order in orders)
{
   foreach (var p in order.Products) // query che estrae prodotti per ordine
   {
   }
}
```

La query filtra gli ordini e per ogni ordine estrae l’id del cliente e gli id dei prodotti coinvolti nell’ordine. In questi casi ci si aspetta che venga eseguita una sola query che estrae i dati utilizzando una `join` SQL. In realtà, il motore esegue prima una query che estrae solo i dati degli ordini (`CustomerId` in questo caso), e poi, quando accediamo in ciclo alla lista dei prodotti per l’ordine corrente, esegue una query per estrarre i prodotti. Questo significa che se la query torna 10 ordini, il codice esegue una query per estrarre i dati dell’ordine e poi una query per ogni ordine per estrarre i prodotti, per un totale di 11 query.

I casi appena mostrati evidenziano come sia sempre bene controllare il codice SQL generato utilizzando strumenti come SqlServer Profiler o sfruttando il logging di Entity Framework Core.

Ottimizzare il fetching

Come abbiamo visto in precedenza, quando recuperiamo gli oggetti, spesso abbiamo bisogno anche di altri oggetti collegati. Nella maggioranza dei casi, la soluzione ideale è quella di recuperare tutti gli oggetti in una singola query. Il problema risiede nel fatto che Entity Framework Core ritorna solo gli oggetti specificati dall’entityset. Questo significa che quando eseguiamo una query sugli ordini, i dettagli vengono ignorati. Fortunatamente possiamo modificare questo comportamento e dire quali dati collegati vogliamo caricare insieme all’entità principale.

Per fare questo dobbiamo utilizzare il metodo `Include` della classe `DbSet`. Questo metodo, in input, accetta una lambda che specifica la proprietà da caricare. Quando dobbiamo caricare proprietà che fanno parte dell’oggetto caricato tramite `Include`, possiamo concatenare il metodo `ThenInclude`, che accetta una lambda che specifica la proprietà da caricare. L’utilizzo di entrambi i metodi è mostrato nel seguente codice.

Esempio 10.16

```
ctx.Orders.Include(o => o.OrderDetails);
ctx.Orders.Include(o => o.OrderDetails).ThenInclude(c => c.Product);
```

Il codice SQL generato dalla prima query mette in join le tabelle `Orders` e `Order Details` per recuperare ordini e dettagli, mentre quello generato dalla seconda query mette in join `Orders` e `Order Details` e `Products` per recuperare ordini, dettagli e i prodotti legati ai dettagli.

Sebbene recuperare immediatamente i dettagli sia ottimale nella maggior parte dei casi, vi sono delle situazioni in cui è meglio recuperare solo gli ordini e accedere ai dettagli sfruttando il lazy loading.

Lazy loading

Tramite questa tecnica effettuiamo una query per recuperare una o più entity principali e ne recuperiamo le entità collegate solo se fisicamente vi accediamo. Per fare un esempio, potremmo dover recuperare tutti gli ordini di un giorno, ma recuperare i dettagli solo di quelli spediti in certi stati. In questo caso recuperiamo gli ordini in una query unica e solo quando accediamo ai dettagli viene effettuata una query da Entity Framework Core (per noi trasparente in quanto accediamo semplicemente alla proprietà `OrderDetails`). Cosa che abbiamo trattato nella precedente sezione, questo significa introdurre il problema del 1+n, poiché si esegue una query per ogni ordine di cui recuperiamo i dettagli, quindi è bene valutare l’uso del lazy loading.

Oltre a questa problematica, per supportare il lazy loading dobbiamo anche apportare modifiche al codice. Per prima cosa, dobbiamo aggiungere tramite NuGet un riferimento all’assembly `Microsoft.EntityFrameworkCore.Proxies`. Successivamente, nella configurazione del contesto dobbiamo aggiungere la chiamata al metodo `UseLazyLoadingProxies`.

Esempio 10.17

```
services.AddDbContext<NorthwindContext>(o =>
{
   o.UseLazyLoadingProxies();
   o.UseSqlServer(”Data Source=(local);Initial Catalog=Northwind;
```

```
      Integrated Security=True”);
});
```

Infine dobbiamo modificare le classi dell’object model, rendendole tutte non `sealed`, e impostando come `virtual` tutte le navigation property. Questa operazione è necessaria in quanto, a run time, Entity Framework Core crea un nuovo tipo (proxy) che eredita dalla nostra classe dell’object model e che sovrascrive getter e setter delle navigation property, al fine di iniettare il codice necessario a effettuare il lazy loading, cioè andare sul database e recuperare i dati. L’Esempio 10.18 mostra come il lazy loading sia trasparente per il nostro codice.

Esempio 10.18

```
var orders = _ctx.Orders.ToList ();
foreach (var order in orders)
{
   if (order.ShipAddress.Country == ”Italy”)
   {
      foreach (var detail in order.OrderDetails)
      {
      }
   }
}
```

Il risultato di questo codice è che per ogni ordine spedito in Italia, noi accediamo ai dettagli ed Entity Framework Core scatena una query in maniera trasparente per noi. Se avessimo caricato tutti i dettagli per tutti gli ordini, avremmo effettuato una sola query al database (anzi due, viste le modalità di querying di Entity Framework Core), ma avremmo caricato moltissimi dati inutili. In questo modo, effettuiamo più query ma carichiamo solo i dati effettivamente utili. La scelta tra una tecnica e l’altra va valutata caso per caso.

> *In questo caso specifico, la soluzione migliore sarebbe stata quella di effettuare una query che estraesse tutti i dettagli degli ordini spediti in Italia, ma lo scopo era solo quello di mostrare il funzionamento del lazy loading.*

Lavorare con i proxy non è sempre la cosa ideale, soprattutto quando si ha a che fare con la reflection. Nei casi in cui non vogliamo usare i proxy ma vogliamo mantenere il lazy loading, possiamo iniettare in un costruttore privato della nostra classe, l’interfaccia `ILazyLoader`, oppure un delegato di tipo `Action<object, string>`, il cui nome deve essere `lazyLoader`. Grazie all’iniezione di questi oggetti, possiamo invocare il caricamento dei dati, scrivendo noi il codice senza dover ricorrere ai proxy e a tutto quello che il loro utilizzo comporta. Per maggiori informazioni su queste tecniche, rimandiamo alla documentazione, disponibile all’indirizzo: http://aspit.co/bnu.

Le possibilità di querying non si fermano qui. Infatti, Entity Framework Core mette a disposizione altre funzionalità che possono tornare utili a seconda degli scenari.

Querying avanzato

Come abbiamo accennato in precedenza, il provider LINQ permette di utilizzare codice SQL scritto a mano e anche di mischiare codice SQL e LINQ in una unica query. Questo è reso possibile dal metodo `FromSql` della classe `DbSet<T>` il quale accetta in input una stringa SQL. Tramite questa stringa possiamo specificare di eseguire una funzione, una stored procedure oppure la clausola from di un comando SELECT, al quale poi possiamo agganciare i metodi di LINQ per formare un’unica query.

Esempio 10.19

```
// Esegue una stored procedure per recuperare ordini
var orders = context.Orders
      .FromSql(”EXECUTE dbo.GetOrdersByCustomer {0}”, customerId)
      .ToList();
// Concatena una query su una Table-Valued Function con LINQ
var orders = context.Orders
      .FromSql(”Select * from dbo.GetOrdersByCustomer {0}”, customerId)
      .Include(o => o.OrderDetails)
      .Where(o => o.ShipAddress.City == ”Rome”)
      .OrderBy(o => o.OrderDate)
.ToList();
```

Nel primo esempio eseguiamo una funzione che restituisce gli ordini, nel secondo caso facciamo una query sul risultato della funzione, specificando che vogliamo estrarre anche i dettagli, che vogliamo prendere solo gli ordini destinati a `Roma` ordinati per data. Questa query viene interamente eseguita sul server grazie al fatto che il provider LINQ riesce a generare un’unica query.

Un’altra funzionalità importante è quella che prevede l’esecuzione delle query asincrone. Entity Framework Core introduce i metodi `ToListAsync`, `ToArrayAsync`, `FirstAsync`, solo per nominarne alcuni, su cui deve essere fatta l’`await` per aspettarne la fine

dell'esecuzione. Va tuttavia specificato che il contesto non è thread safe, quindi l'asincronia non può essere sfruttata per eseguire più query in parallelo.

L'ultima funzionalità da tenere a mente è la capacità di esprimere dei filtri a livello globale per gli entityset. Durante la fase di mapping possiamo specificare un filtro su una classe dell'object model ed Entity Framework Core applicherà sempre questo filtro. Questa tecnica torna utile quando abbiamo un database multi-tenant e vogliamo estrarre solo i record associati al tenant dell'utente loggato o quando usiamo la cancellazione logica dei record e vogliamo estrarre solo quelli non cancellati. In quest'ultimo caso, applicare un filtro globale che esclude quelli non cancellati ci consente di non dover scrivere il filtro in ogni query.

Esempio 10.20

```
//mapping con filtro globale
modelBuilder.Customers<Orders>().HasQueryFilter(c => !c.Deleted);

//query
var orders = await context.Customers.ToListAsync();
```

Nei casi in cui non vogliamo che i filtri siano onorati in una specifica query, dobbiamo concatenare alla query la chiamata al metodo `IgnoreQueryFilters`.

Come si è visto nel corso della sezione, la scrittura di query in Entity Framework Core è estremamente potente, nonostante i problemi di gioventù. Passiamo ora a vedere, invece, come persistere i dati sul database.

Salvare i dati sul database

Quando un oggetto viene recuperato dal database, viene anche memorizzato all'interno del contesto, in un componente chiamato change tracker.

Sono due i motivi per cui questo comportamento è necessario. Innanzitutto, ogni volta che eseguiamo una query, il contesto (che è responsabile della creazione fisica degli oggetti) verifica che non esista già un oggetto corrispondente (ovvero con la stessa chiave primaria) nel change tracker. In caso affermativo, il record proveniente dal database viene scartato e il relativo oggetto nel change tracker viene restituito. In caso negativo, viene creato il nuovo oggetto, messo nel change tracker e, infine, restituito all'applicazione. Questo garantisce che vi sia un solo oggetto a rappresentare lo stesso dato sul database. Inoltre, il contesto deve mantenere traccia di tutte le modifiche fatte agli oggetti, così che poi queste possano essere riportate sul database. Il fatto di avere gli oggetti in memoria semplifica questo compito. Salvare i dati sul database significa persistere la cancellazione, l'inserimento e le modifiche delle entità. Questo processo viene scatenato attraverso la chiamata al metodo `SaveChanges` (o la sua controparte asincrona `SaveChangesAsync`) del contesto. Questo metodo non fa altro che scorrere gli oggetti modificati memorizzati nel contesto, iniziare una transazione, costruire ed eseguire il codice SQL per la persistenza e, infine, eseguire il commit o il rollback a seconda che vada tutto bene oppure no.

> *Se abbiamo più chiamate al metodo SaveChanges e vogliamo che queste siano eseguite in un'unica transazione, dobbiamo usare la classe TransactionScope. Poiché il supporto per TransactionScope è stato introdotto in .NET Core 2.1 e poiché va abilitato nei provider specifici per database, è possibile che alcuni provider inizialmente non lo supportino. È tuttavia probabile che in tempi brevi tutti i provider supportino queste API. Per sicurezza è bene consultare la documentazione del produttore.*

Per poter generare il codice SQL corretto, il contesto deve conoscere lo stato di ogni singolo oggetto quando effettua il salvataggio. Un oggetto può essere in quattro stati:

- `Unchanged`: l'oggetto non è stato modificato. Il processo di persistenza non scatena alcun comando per gli oggetti in questo stato.
- `Modified`: qualche proprietà semplice dell'oggetto è stata modificata. Il processo di persistenza genera un comando SQL di UPDATE per ogni oggetto in questo stato, includendo solo le colonne modificate.
- `Added`: l'oggetto è stato marcato per l'aggiunta sul database. Il processo di persistenza genera un comando SQL di INSERT per ogni oggetto in questo stato.
- `Deleted`: l'oggetto è stato marcato per la cancellazione dal database. Il processo di persistenza genera un comando SQL di DELETE per ogni oggetto in questo stato.

Quando un oggetto viene creato da una query, il suo stato è `Unchanged`. Nel momento in cui andiamo a modificare una proprietà semplice o una complessa, lo stato passa automaticamente a `Modified`. `Added` e `Deleted` si differenziano dagli stati precedenti, in quanto devono essere impostati manualmente. Per settare lo stato di un oggetto su `Added`, dobbiamo utilizzare il metodo `Add` della

classe DbSet<T>, mentre per impostare un oggetto su Deleted, dobbiamo invocare Remove. Analizziamo ora queste casistiche in dettaglio.

Persistere un nuovo oggetto

In ogni applicazione che gestisce il ciclo completo di un'entità, il primo passo è l'inserimento. Come detto poco sopra, per persistere un nuovo oggetto basta utilizzare il metodo Add della classe DbSet<T>, passando in input l'oggetto da persistere. L'effetto è quello di aggiungere l'oggetto al contesto nello stato di Added.

Esempio 10.21

```
using (var ctx = new NorthwindEntities())
{
   var c = new Customer { /*Imposta proprietà customer*/ };
   ctx.Customers.Add(c);
   ctx.SaveChanges();
}
```

Come possiamo notare nell'esempio, inserire un nuovo cliente è estremamente semplice. Quando si deve inserire un ordine, invece, la situazione cambia leggermente, poiché un ordine contiene riferimenti ai dettagli e anche un riferimento al cliente.

In tal caso, il metodo Add si comporta in questo modo: imposta l'oggetto passato in input in stato di Added, poi scorre tutte le sue navigation property, così da aggiungere gli oggetti collegati al contesto, mettendoli nello stato di Added. In questo modo, possiamo richiamare una volta soltanto il metodo Add, passando l'ordine, con il vantaggio che tanto i dettagli quanto il cliente verranno aggiunti al contesto in fase di inserimento.

Tuttavia, se per i dettagli questo rappresenta l'approccio corretto, per il cliente non possiamo dire altrettanto, visto che, in realtà, non deve essere inserito nuovamente. In questi casi dobbiamo ricorrere ai metodi Attach e Update. Entrambi i metodi, come Add, prendono un oggetto in input e ne scorrono tutte le navigation property per attaccarle al contesto, ma cambiano il modo di impostare lo stato. Per ogni oggetto, se il valore della chiave primaria è uguale al valore di default del suo tipo (0 per interi, null per stringhe e così via), allora il relativo stato viene impostato su Added, altrimenti viene impostato su Unchanged o Modified, a seconda del metodo usato.

Esempio 10.22

```
using (var ctx = new NorthwindEntities())
{
   var o = new Order {
      Customer = new Customer { CustomerId = "ALFKI" },
      //Imposta altre proprietà ordine ma non la chiave perché ordine nuovo
   };
   o.OrderDetails.Add(new OrderDetail {
      //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
   });
   o.OrderDetails.Add(new OrderDetail {
      //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
   });
   ctx.Orders.Attach(o);
   ctx.SaveChanges();
}
```

Nel caso dell'esempio 10.22, poiché non impostiamo la chiave né dell'ordine né dei dettagli, questi vengono aggiunti al contesto in stato di Added, mentre il cliente viene aggiunto al contesto in stato di Unchanged, visto che la sua chiave è impostata.

Ora che abbiamo imparato a inserire nuovi oggetti, passiamo ad analizzarne la rispettiva modifica.

Persistere le modifiche a un oggetto

Per modificare un cliente, basta leggerlo dal database, modificarne le proprietà e invocare il metodo SaveChanges/SaveChangesAsync come mostrato nel prossimo esempio.

Esempio 10.23

```
using (var ctx = new NorthwindEntities())
{
   var cust = ctx.Customers.Find("ALFKI");
   cust.Address.Address = "Piazza del popolo 1";
   ctx.SaveChanges();
}
```

Questa modalità di aggiornamento viene definita come connessa, poiché l'oggetto è modificato mentre il contesto che lo ha istanziato è ancora in vita.

Tuttavia non è sempre così. Prendiamo, per esempio, un Web Service che metta a disposizione un metodo che torna i dati di un cliente e un altro che accetti un cliente e lo salvi sul database. La cosa corretta, in questi casi, è creare un contesto diverso in ogni metodo. Questo significa che non c'è nessun tracciamento delle modifiche fatte sul client e quindi il secondo contesto non può sapere cosa è cambiato. Quando incontriamo questo tipo di problema, si parla di modalità disconnessa, in quanto le modifiche all'oggetto sono fatte fuori dal contesto che lo ha creato.

In questi casi abbiamo a disposizione due modalità per risolvere il problema. Con la prima effettuiamo nuovamente la query per recuperare il cliente e impostiamo le proprietà con i dati che vengono passati in input al metodo. In questo caso il contesto riesce a tracciare le modifiche, quindi il codice è identico a quello visto nell'Esempio 10.24.

La seconda consiste nell'*attaccare* l'entità modificata al contesto, marcandola come `Modified` tramite il metodo `Update` già analizzato in precedenza. L'Esempio 10.24 mostra come usare questa tecnica.

Esempio 10.24

```
//Metodo che del servizio torna il cliente
using (var ctx = new NorthwindEntities())
{
   return ctx.Customers.Find("STEMO");
}

//Il client aggiorna il cliente e chiama il servizio di aggiornamento
Cust.Address.Address = "Piazza Venezia 10";
UpdateCustomer(Cust);

//Il servizio aggiorna il cliente
using (var ctx = new NorthwindEntities())
{
   ctx.Customers.Update(modifiedCustomer);
   ctx.SaveChanges();
}
```

Quando si parla di ordini e dettagli o, più in generale, di classi che hanno navigation properties, è importante sottolineare una cosa. Se aggiungiamo, modifichiamo o cancelliamo un dettaglio in modalità disconnessa, anche impostando l'ordine come modificato non otteniamo l'aggiornamento dei dettagli. Per eseguire un corretto aggiornamento, dobbiamo fare una comparazione manuale tra gli oggetti presenti sul database e quelli presenti nell'oggetto modificato e cambiare manualmente lo stato di ogni dettaglio. Dopo la modifica, è arrivato il momento di passare alla cancellazione dei dati.

Cancellare un oggetto dal database

L'ultima operazione di aggiornamento sul database è la cancellazione. Cancellare un oggetto è estremamente semplice, in quanto basta invocare il metodo `Remove` della classe `DbSet<T>`, passando in input l'oggetto. C'è comunque una particolarità molto importante da tenere presente. L'oggetto da cancellare deve essere attaccato al contesto. Questo significa che, anche in caso di cancellazione, abbiamo una modalità connessa e una disconnessa.

Nella modalità connessa possiamo semplicemente eseguire una query per recuperare l'oggetto e invocare poi `Remove` (Esempio 10.25).

Esempio 10.25

```
using (var ctx = new NorthwindEntities())
{
   var cust = ctx.Customers.Find("STEMO");
   ctx.Customers.Remove(cust);
}
```

Anche nella modalità disconnessa abbiamo due tipi di scelta: recuperare l'oggetto dal database e invocare `Remove` (stesso codice dell'esempio qui sopra) oppure attaccare l'oggetto al contesto e poi chiamare il metodo `Remove` come nel prossimo esempio.

Esempio 10.26

```
using (var ctx = new NorthwindEntities())
{
   ctx.Customers.Attach(custToDelete);
   ctx.Customers.Remove(custToDelete);
}
```

Come abbiamo detto, gli oggetti attaccati al contesto vengono salvati nel change tracker. Questo componente espone alcuni metodi che possono essere utili per manipolare gli oggetti al suo interno.

Manipolare gli oggetti nel change tracker

Il contesto espone il change tracker tramite la proprietà `ChangeTracker` di tipo `ChangeTracker`. Il metodo più importante di questa classe è `Entries`, che ritorna una lista di oggetti di tipo `EntityEntry`. Ogni istanza di `EntityEntry` (detta anche entry) contiene i dati di un oggetto attaccato al contesto, li espone attraverso le sue proprietà e permette di manipolarli attraverso i suoi metodi. Quelli principali sono esposti nella Tabella 10.4.

Tabella 10.4 – Membri di EntityEntry.

Membro	Scopo
`Entity`	Proprietà che restituisce l'entity associata all'entry
`State`	Proprietà che restituisce e imposta lo stato dell'entity
`IsKeySet`	Specifica se la chiave dell'entity è impostata
`Metadata`	Restituisce i dati di mapping dell'entity
`OriginalValues`	Restituisce i valori dell'entity com'era quando è stata attaccata al contesto (recuperandola dal database o usando uno dei metodi per attaccarla al contesto)
`CurrentValues`	Restituisce i valori attuali dell'entity
`Reload/ReloadAsync`	Ricarica l'entity prendendo i valori dal database

Come abbiamo detto, `Entries` restituisce una lista di entry. Questo significa che possiamo filtrare le entry per recuperarne una specifica e modificarne lo stato, oppure tutte quelle in uno stato di `Added` per loggare le operazioni di inserimento o altro ancora.

Quando abbiamo un riferimento a una entity e vogliamo recuperare l'entry associata, possiamo usare il metodo `Entry` del contesto che accetta in input la entity e restituisce l'entry.

Aggiornare i dati con Entity Framework Core è tutt'altro che complesso, quindi non necessita ulteriori approfondimenti. Nella prossima sezione, invece, vedremo brevemente altre caratteristiche che possono tornare utili durante lo sviluppo.

Funzionalità aggiuntive di Entity Framework Core

Ovviamente, in questo capitolo abbiamo trattato solo gli argomenti principali che ci permettono di sviluppare con Entity Framework Core. Infatti, esistono molte altre funzionalità che questo framework ci mette a disposizione. In questa sezione elenchiamo quelle più comuni:

- Concorrenza: Entity Framework Core gestisce la concorrenza ottimistica. L'unica cosa che dobbiamo fare per abilitare la concorrenza è indicare nel mapping quali campi devono essere controllati.

- Code-First migration: permette di creare il database a design-time, partendo dalle classi del modello a oggetti. Inoltre, permette di modificare il database al cambio del codice delle classi mappate.

- Validazione: poiché nel mapping specifichiamo molte informazioni, come per esempio la lunghezza di una proprietà di tipo `String`, il contesto prima di persistere le notifiche verifica che le proprietà siano conformi al loro mapping, evitando così di arrivare al database con dati non validi.

- Logging: Entity Framework Core ha un motore di logging integrato, dove possiamo intercettare l'esecuzione dei comandi e generare un log.

- Mapping avanzato: Entity Framework Core permette di mappare i campi di una tabella su più classi, di mappare l'ereditarietà secondo il modello TPH, di mappare campi della tabella verso membri privati di una classe e di specificare chiavi univoche alternative.

Conclusioni

In questo capitolo abbiamo parlato delle caratteristiche principali di Entity Framework Core, così da poter cominciare a utilizzare questo O/RM. Ora siamo in grado di costruire e modificare un modello sia scrivendo a mano l'object model, il contesto e il mapping, sia sfruttando i tool che Entity Framework Core ci mette a disposizione per generare le stesse cose partendo dal database. Abbiamo visto come recuperare i dati dal database sfruttando il provider LINQ e come persistere sia oggetti semplici sia grafi complessi con poche righe di codice.

Tuttavia c'è molto di più. Come abbiamo visto nell'ultima sezione del capitolo, Entity Framework Core offre una serie di funzionalità che lo rendono uno strumento valido per l'accesso ai dati. Questo, unito alla capacità di girare anche su piattaforme Linux e MacOS, apre scenari finora impossibili da immaginare, il che fa capire come Microsoft stia investendo molto su questa tecnologia, che raffigura il presente e il futuro dell'accesso ai dati.

Tuttavia, Entity Framework Core è ancora molto giovane e necessita miglioramenti di stabilità, performance e funzionalità. In alcuni scenari, queste mancanze non sono un problema, mentre in altri rappresentano un ostacolo insormontabile. Prima di scegliere Entity Framework Core è sempre bene ponderare la scelta in base alle proprie esigenze.

Ora che abbiamo visto come accedere ai dati residenti su un database, nel prossimo capitolo vedremo come pubblicarli attraverso servizi REST.

11
Sviluppare servizi RESTful con ASP.NET Core WebAPI

Nel corso del libro abbiamo introdotto le caratteristiche del pattern MVC supportato da ASP.NET Core, parlando di Controller, Dependency Injection, routing e view. Nel capitolo precedente, invece, sono stati introdotti concetti molto legati ai dati per spiegare com'è possibile recuperarli ed elaborarli utilizzando O/RM come Entity Framework Core.

Date le basi già affrontate, all'interno di questo capitolo tratteremo una sorta di variazione sul tema: non è detto che i dati debbano sempre essere messi in comunicazione e mostrati all'interno di una view ma, al contrario, spesso vanno esposti, per realizzare servizi, che potranno anche essere riutilizzati, anche se magari solo in parte, nelle view di MVC.

Proprio per questi motivi, all'interno di questo capitolo introdurremo i concetti legati a WebAPI e REST API, ne capiremo le differenze e cercheremo di evidenziare i vantaggi nell'utilizzare sistemi di self-discovery per riuscire a sviluppare in modi separati client e server.

I servizi RESTful

REST (REpresentational State Transfer), al contrario di quello che si potrebbe credere, non impone una definizione di come deve essere costruita una API ma indica il modo in cui gli endpoint si comportano in funzione dell'URI specificato: l'utilizzo di JSON o XML come formato di risposta non è definito, perché REST non si occupa di protocolli. A voler essere precisi, il protocollo HTTP non fa parte di REST, sebbene sia complesso immaginarsi scenari di API che non fanno uso di HTTP.

> *REST non è uno standard, ma uno stile architetturale con cui disegnare degli endpoint, in modo che sia più facile per tutti capire come invocarli.*

La definizione di REST passa attraverso questi principi fondamentali:

- Client (il consumatore del servizio) e server (il produttore) sono architetturalmente separati e devono evolvere in modo indipendente: non è necessario che il client si preoccupi di capire come vengono recuperati i dati lato server, così come per il server non è necessario capire come verranno utilizzati i dati nel client.

- Il servizio deve essere stateless: lo stato viene gestito richiesta per richiesta ed è per questo che viene garantita la scalabilità.

- I dati devono essere mantenibili: ogni risposta mandata dal server deve specificare la durata di validità dei dati (cache) e deve possibilmente essere immediata, ovvero composta architetturalmente da meno strati possibili (uno, nel caso ideale).

- Ogni risorsa deve essere identificata in modo univoco: le risorse devono essere separate dalla loro rappresentazione (XML o JSON che sia).

- Le risposte devono contenere informazioni necessarie per fare il discovery e spiegare al client come utilizzare le API (HATEOAS).

La maggior parte delle API di cui facciamo uso tutti i giorni, probabilmente senza accorgercene, non fa uso corretto di REST, ma questo non implica che non siano API valide o che non funzionino nel modo corretto. Sebbene sia un pattern architetturale, seguire i principi alla base di REST implica applicare dei suggerimenti a livello di design che non tutti gli sviluppatori e gli architetti sono disposti ad accettare, oppure sono in grado di seguire. Per individuare il grado di aderenza a REST, esiste un modello, creato da Leonard Richardson e visibile più nel dettaglio su http://aspit.co/bnp, che spiega nel dettaglio che cosa può essere considerato davvero REST e che dà per scontato l'uso di HTTP come protocollo di comunicazione:

- `Livello 0`: viene utilizzato lo stesso endpoint (per esempio www.miosito.com/api) per fare tutte le chiamate ai vari servizi esposti, poiché gestiti internamente dall'endpoint. Diventa molto difficile per il client capire che cosa succede e fare il discovery di altre risorse oppure, più semplicemente, capire se deve aspettarsi dei dati in risposta, visto che viene utilizzato sempre lo stesso verb HTTP.

- `Livello 1`: viene specificato un endpoint differente per ogni risorsa, per esempio `/api/` customers per lavorare con oggetti di tipo `Customers`, oppure `/api/customers/1` per recuperare uno specifico customer. Ancora una volta non si fa uso dei verb dell'HTTP e quindi, tecnicamente, non si sta aderendo ad alcuno standard.

- `Livello 2`: ogni link viene invocato con uno specifico verb, per avere una determinata funzionalità: useremo, quindi, `GET` per leggere i dati, `POST` per fare inserimenti e così via. Inoltre, ogni risposta inviata al client deve contenere il corretto status

code che identifica l'esito della chiamata (per esempio 200 (OK) per una chiamata andata a buon fine, 201 (Created) per una chiamata andata a buon fine e con un oggetto creato in POST).

- `Livello 3`: indica il supporto per HATEOAS e in ogni risposta deve essere specificato come raggiungere altri luoghi se necessario (per esempio, dopo la creazione di un `Customer`, sarebbe opportuno avere una location per poterlo raggiungere e visualizzare).

In tutte queste specifiche viene solamente menzionato il fatto di avere endpoint differenti per ogni chiamata, ma non viene definita in alcun modo una convenzione dei nomi, pertanto viene rimandato tutto al buon senso e alla capacità del designer delle API.

Vengono comunque utilizzate, in genere, diverse pratiche, tra cui:

- L'uso di sostantivi al posto delle azioni specifiche negli URL: una chiamata a `/api/getcustomers` viene considerata sbagliata e dovrebbe essere sostituita con una chiamata GET ad `/api/customers`.

- Predicibilità: per recuperare un oggetto specifico, l'URL deve essere piuttosto immediato, quindi per recuperare un oggetto customer, viene considerato opportuno costruire un URL come `/api/customers/id` piuttosto che `/api/id/customers` perché l'id all'inizio dell'URL potrebbe essere conteso fra più risorse (libri, prodotti, clienti ecc.).

- L'uso di filtri in query string: tutto ciò che non è definibile come risorsa, quindi filtri e ordinamenti, dovrebbe fare parte di parametri della query string, non del path e, pertanto, un ordinamento potrebbe essere fatto come `/api/customers?orderby= firstName` anziché con `/api/customers/orderby/firstName`.

Nonostante l'introduzione di tutte queste regole, è importante riconoscere che l'uso di ASP.NET Core, come vedremo all'interno del capitolo, ma anche di una qualsiasi altra tecnologia, non permette la creazione di API "certificate" RESTful out-of-the-box: essendo un toolkit flessibile, fornisce tutti gli strumenti necessari per poterle realizzare con facilità. Resta compito degli architetti software progettare un sistema che sia in grado di lavorare secondo i principi definiti in precedenza.

Dando uno sguardo diretto ad ASP.NET Core e all'implementazione delle WebAPI, notiamo già una grossa differenza rispetto al passato: non ci sono più due framework separati, poiché la parte di ASP.NET MVC e quella di ASP.NET WebAPI sono integrate all'interno di un unico framework. ASP. NET Core MVC, infatti, è pensato e realizzato come un unico framework in grado di realizzare endpoint, a prescindere dalla tipologia di risposta che questi avranno.

Implementare un endpoint per le API con ASP.NET Core

La Figura 11.1 mostra l'infrastruttura condivisa tra ASP.NET Core MVC e ASP.NET Core Web API. Possiamo notare come, di fatto, il controller sia uno, a prescindere dal tipo di ritorno, e la differenza lo faccia il modo in cui gli sviluppatori vanno a scrivere effettivamente la action.

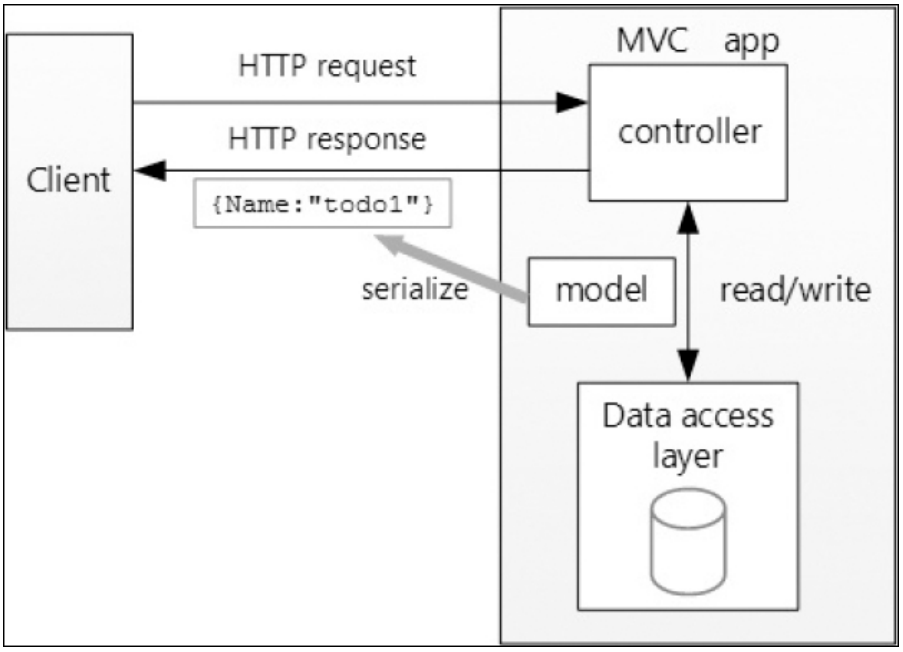


Figura 11.1 – Architettura di una ASP.NET Core WebAPI classica.

Il flusso è semplice: il client manda la richiesta HTTP verso il server (che possiamo simulare con un tool come Postman o curl), il server intercetta la chiamata tramite un controller e le relative action, passando tramite diversi layer di servizi e, eventualmente, interrogando il database, per poi ritornare la risposta tramite un modello serializzato come risposta.

Nell'esempio che vedremo illustrato di seguito, non andremo a fare uso di Entity Framework Core e di un database vero e proprio, perché non è strettamente necessario per illustrare il funzionamento delle API: tutti i dati generati ed elaborati saranno fittizi ma comunque utili per spiegarne il flusso. In particolare, è già stata preparata la classe `Customer`, che identifica un set di clienti che possono acquistare una serie di prodotti di tipo `Product`, mentre tutte le operazioni tipiche, ovvero l'aggiunta, la rimozione e la lettura, passano attraverso una classe chiamata `CustomerRepository` (e la relativa interfaccia `ICustomerRepository`) che viene iniettata all'interno del costruttore come servizio transient.

Il codice è disponibile nell'Esempio 11.1.

Esempio 11.1

```
public class CustomersController : Controller
 {
   private readonly ICustomerRepository customerRepository;
   public CustomersController(ICustomerRepository customerRepository)
   {
      this.customerRepository = customerRepository;
   }
}
```

Dal precedente codice possiamo già notare come il controller appena creato erediti a tutti gli effetti dalla classe `Controller`, al contrario di quanto avveniva con ASP.NET WebAPI, in cui i controller ereditavano da `WebApiController`, rendendo i controller di ASP.NET MVC incompatibili con quelli di ASP.NET Web API.

Aggiungere la prima API che va a leggere e ritornare l'elenco di tutti i clienti è facile tanto quanto scrivere un metodo, come possiamo vedere nell'Esempio 11.2.

Esempio 11.2

```
public JsonResult GetCustomers()
{
   var customers = customerRepository.GetCustomers();
   // restituiamo un risultato di tipo JSON
   return new JsonResult(customers);
}
```

In questo caso specifico, viene richiesto al repository che si occupa della gestione dei clienti di ritornare l'elenco degli stessi e, quindi, viene fatta una serializzazione dei dati in oggetti con il formato JSON, attraverso la classe `JsonResult`, che rimanda tutti i dati al client che ha effettuato la richiesta. Se proviamo ad avviare il server e a eseguire la richiesta con Postman, noteremo però un errore 404 in risposta alla chiamata su `/api/customers`, come viene illustrato nella Figura 11.2.

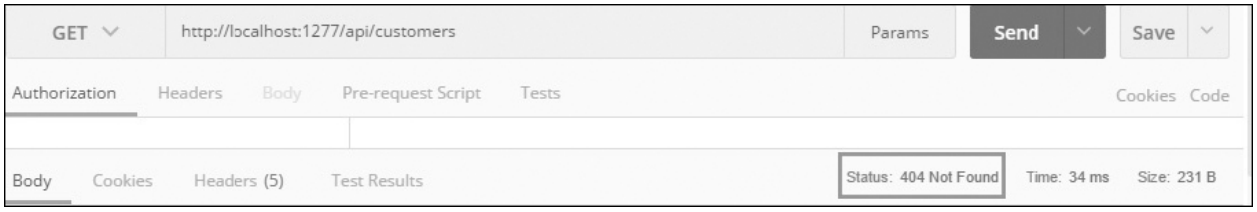


Figura 11.2 – La chiamata `GET` verso un endpoint non configurato produce un errore.

Questo errore è del tutto normale, perché non è stato ancora comunicato ad ASP.NET Core che vogliamo che l'endpoint risponda su un determinato URL.

Per farlo, è necessario fare uso del routing, così da mappare la action del controller a un URL specifico: nonostante esistano due tipi di routing, convention-based e attribute-based, Microsoft consiglia di utilizzare il primo per la definizione delle rotte sulla parte "front-end" di MVC, ovvero quella che viene poi utilizzata dal sito web, mentre consiglia di fare uso degli attributi per lavorare con le API.

Per rispondere al link corretto che è stato predisposto in Postman, è necessario modificare gli Esempi 11.1 e 11.2 così che assumano una forma come quella mostrata nell'Esempio 11.3.

Esempio 11.3

```
[Route("api/customers")]
public class CustomersController : Controller
{
```

```
    //...
    [HttpGet]
    public JsonResult GetCustomers()
    {
        var customers = customerRepository.GetCustomers();
        return new JsonResult(customers);
    }
}
```

Nell'Esempio 11.3 è stata registrata una route che risponde all'indirizzo `/api/customers` e, grazie all'uso dell'attributo `HttpGet`, la risposta sarà servita solo se il verb HTTP con la quale viene invocato l'endpoint è quello `GET`.

In realtà, possiamo fare uso delle convenzioni di ASP.NET Core per modificare il valore dell'attributo `Route` in api/[controller]: così facendo, il nome del controller viene passato in automatico all'attributo che gestisce la route ma, in caso di refactoring, tutte le API che vengono esposte dal controller corrente cambierebbero l'endpoint. Si tratta di una scelta personale, anche se di solito è fortemente sconsigliato lasciare l'endpoint dinamico, a meno che venga usato in concomitanza con un sistema di API versioning tipo Swagger, di cui vedremo i dettagli nel corso del prossimo capitolo.

Una volta configurata correttamente la route, il risultato ottenuto sarà simile a quello illustrato nella Figura 11.3.

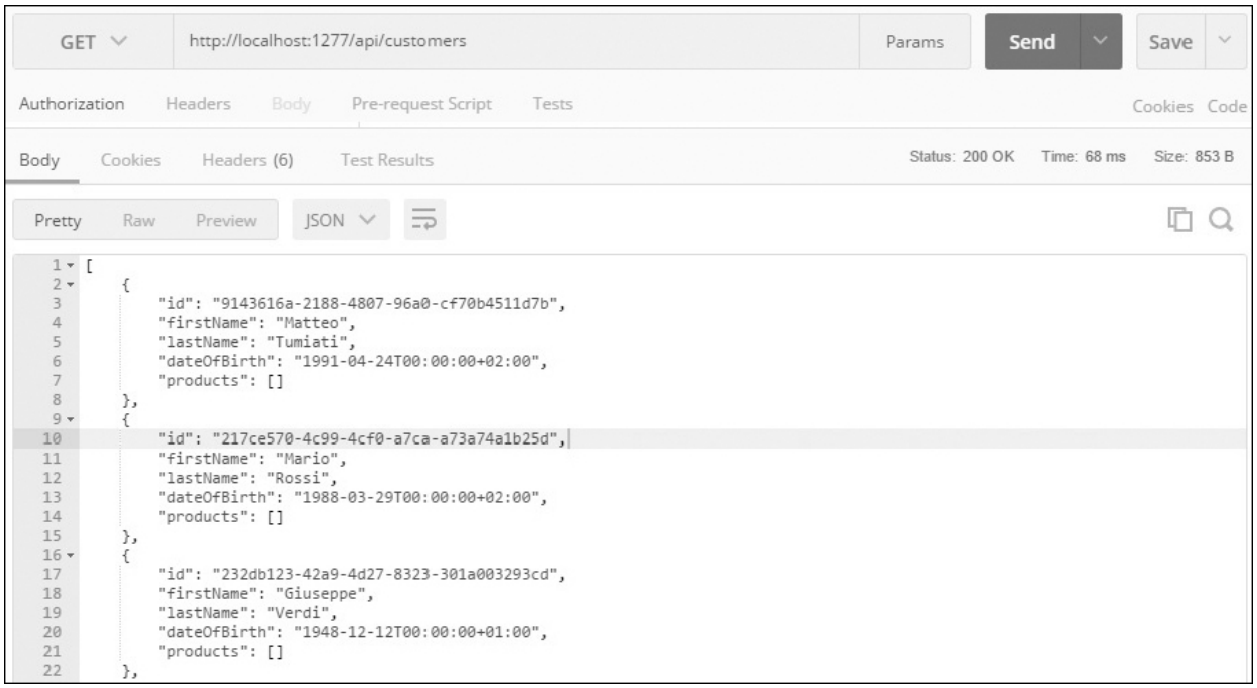


Figura 11.3 – Ecco come si presenta la risposta a una chiamata con il verb `GET` con l'elenco di oggetti di tipo `Customer`.

In base alla risposta ottenuta, possiamo osservare che, poiché la funzione sta mandando in risposta un oggetto di tipo JSON, viene mantenuta la notazione corretta delle proprietà in camelCase. Inoltre, anche se forse meno ovvio, stiamo mandando in risposta un oggetto identico a quello che è stato potenzialmente estratto a partire dal database e, pertanto, così facendo vengono esposte proprietà che potrebbero non essere di interesse pubblico o rientrare negli scopi per cui viene definita questa API: è importante, infatti, avere una separazione, non solo per proteggere il database da eventuali attacchi, ma anche perché, in caso di aggiornamenti della struttura dati, l'API pubblicata manterrebbe un risultato identico, senza il bisogno di agire sul versioning. Nell'esempio proposto, potrebbe essere utile, infatti, ritornare una sola proprietà con il nome completo, anziché esporre nome e cognome come proprietà separate, oppure potrebbe fare comodo mostrare direttamente l'età, anziché mostrare la data di nascita e lasciare il calcolo al client.

Nell'Esempio 11.4, pertanto, facciamo uso di un DTO (Data Transfer Object), cioè di un modello predisposto esattamente allo scopo di trasferire dati all'interno di servizi.

Esempio 11.4

```
[HttpGet]
public JsonResult GetCustomers()
{
    var customers = customerRepository.GetCustomers();
    var customersForRead = customers.Select(customer => new CustomerForRead
    {
        Id = customer.Id,
```

```
      Name = $"{customer.FirstName} {customer.LastName}",
      Age = customer.DateOfBirth.GetAge()
   });
   return new JsonResult(customersForRead);
}
```

L'Esempio 11.4 dimostra come, tramite una semplice query di LINQ, sia possibile creare una nuova lista di oggetti di tipo `CustomerForRead`, in cui, al contrario della classe `Customer`, sono state esposte solo le proprietà da trasportare, come `Id`, utile per recuperare eventualmente l'utente singolo in un momento successivo, `Name` che espone il nome completo e `Age` che rappresenta l'età a partire dalla data di nascita grazie all'uso di un extension method costruito ad-hoc per questo esempio. L'esempio in questione funziona bene nel caso in cui ci siano da modificare poche proprietà ma, in caso di oggetti complessi e di API che espongano più volte lo stesso modello, potrebbe diventare noioso riscrivere più e più volte lo stesso codice, e, inoltre, si correrebbe il rischio di commettere errori.

Per nostra fortuna, ci sono utility come AutoMapper, una libreria disponibile su NuGet che si occupa di object-mapping e cioè, tramite l'utilizzo di convenzioni, è in grado di convertire un oggetto in ingresso in un oggetto in uscita che ha proprietà simili, oppure proprietà calcolate secondo le direttive impostate a livello centrale. La libreria, come tutta la relativa documentazione, è open source e disponibile su GitHub all'indirizzo `http://aspit.co/bnq`.

Una volta installata la libreria, è necessario applicare un po' di configurazione per istruire il motore di mapping all'interno del metodo `Configure` della classe `Startup`. L'Esempio 11.5 mostra come fare.

Esempio 11.5

```
AutoMapper.Mapper.Initialize(config =>
{
   config.CreateMap<Customer, CustomerForRead>()
      .ForMember(o => o.Name, i => i.MapFrom(m => $"{m.FirstName} {m.LastName}"))
      .ForMember(o => o.Age, i => i.MapFrom(m => m.DateOfBirth.GetAge()));
});
```

A questo punto, poiché il mapping viene fatto in tutti i casi in cui dovesse essere necessario dalla libreria, registrata globalmente, nella funzione `GetCustomers` è possibile eliminare la query LINQ che si occupa della trasformazione dei dati, sostituendola con una chiamata al motore di mapping, come è visibile nell'Esempio 11.6.

Esempio 11.6

```
[HttpGet]
public JsonResult GetCustomers()
{
   var customers = customerRepository.GetCustomers();
   var customersForRead =
                        Mapper.Map<IEnumerable<CustomerForRead>>(customers);
   return new JsonResult(customersForRead);
}
```

Il risultato ottenuto sarà simile a quello illustrato nella Figura 11.4.

```
1  [
2      {
3          "id": "ec3d723c-fca5-455a-bfb2-13be9e54caa1",
4          "name": "Matteo Tumiati",
5          "age": 27
6      },
7      {
8          "id": "1cd20b70-2028-41c7-b6b5-da7093c65245",
9          "name": "Mario Rossi",
10         "age": 30
11     },
12     {
13         "id": "5efc3f67-3d42-42b7-b4c1-ee4ef9e2a364",
14         "name": "Giuseppe Verdi",
15         "age": 69
16     },
17     {
18         "id": "bd627a3a-7ae4-4d5c-bb43-dd98394e75b5",
19         "name": "Marco Bianchi",
20         "age": 43
21     }
22 ]
```

Figura 11.4 – Manipolazione dei dati in risposta grazie al mapping con AutoMapper sulla classe DTO.

Poiché il mapping viene ora fatto verso la classe DTO `CustomerForRead`, come si nota nella Figura 11.4, i dati ottenuti in risposta non sono più i dati originali esposti dalla classe `Customer`, come per esempio la data di nascita, ma sono i dati manipolati da AutoMapper che va a esporre la sola età e il nome completo anziché i due valori come campi separati.

Quando si parla di recupero dei dati e di servizi internet, però, è sempre bene prestare attenzione al fatto che tutto può andare storto, dalla connessione internet mancata fra due servizi che si parlano, all'irraggiungibilità di un database, fino a un errore nel codice. Finora abbiamo dato per scontato che il servizio funzioni sempre e che produca una risposta corretta ma, nel caso in cui si verifichino degli errori, è bene avvisare il client così che, eventualmente, possa riprovare la chiamata per avere la risposta che si aspetta.

Status code e gestione degli errori

Una volta ottenuto l'elenco dei clienti, il prossimo passo è quello di realizzare un'API che esponga una singola entità. La costruzione di questa API è piuttosto semplice e la logica continua a dipendere dall'architettura scelta per il progetto, come possiamo notare nell'Esempio 11.7.

Esempio 11.7

```
[HttpGet("{id}")]
public JsonResult GetCustomer(Guid id)
{
   var customer = customerRepository.GetCustomer(id);
   var customerForRead = Mapper.Map<CustomerForRead>(customer);
   return new JsonResult(customerForRead);
}
```

Per ottenere una singola entità, viene usata quella che è la sua chiave primaria, in modo tale che l'oggetto venga identificato in modo univoco e, come viene dimostrato nell'Esempio 11.7, la firma del metodo accetta in ingresso un parametro di tipo `Guid`, il cui nome corrisponde a quello della chiave primaria `Id`. Il nome della proprietà deve essere mantenuto identico a quello esplicitato nella route dell'attributo `HttpGet`, in modo che il framework sia in grado di ricostruirlo al momento della richiesta: in caso non ci fosse una corrispondenza diretta, infatti, le proprietà del metodo assumeranno il loro valore di default, variabile a seconda del tipo: questo comportamento è del tutto simile a quanto abbiamo già visto nei capitoli precedenti ed è legato al funzionamento del routing.

La richiesta, in questo caso, verrà mappata sull'endpoint `/api/customers/{id}` e verrà restituito l'oggetto corrispondente all'ID selezionato. Qualora non ci sia un elemento corrispondente, a fronte di una richiesta valida, la risposta che si otterrà è forse più naturale ma meno funzionale del previsto e la possiamo vedere nella Figura 11.5.

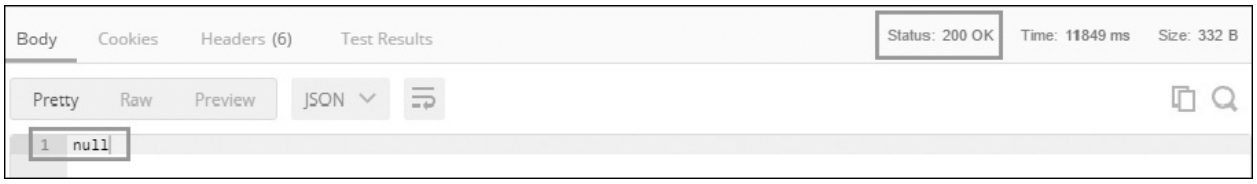


Figura 11.5 – In caso di mancanza di dati, anziché una risposta errata, si ottiene comunque una risposta di tipo 200 (OK).

Osservando nel dettaglio, l'oggetto inviato in risposta è di tipo `CustomerForRead`, per cui viene inviata una risposta di tipo `null`, perché una corrispondenza nello storage non è stata trovata. Per questo motivo, lo status code inviato con la risposta è 200 (OK), che in questo caso è formalmente sbagliato, poiché la richiesta non è andata a buon fine.

Parte di un buon design di servizi REST è il fare uso del corretto status code HTTP. Questo è importante, perché i client che andranno a leggere e a elaborare i dati ottenuti dalle API esposte potranno capire in automatico se le richieste sono state eseguite correttamente oppure se si sono verificati errori e, in questo caso specifico, capirne la tipologia, per ritentare eventualmente la chiamata all'API corrispondente.

Gli status code fanno parte dello standard HTTP e si dividono principalmente in cinque macro-categorie:

- `100`: sono codici informativi aggiunti dopo l'introduzione dello standard HTTP;
- `200`: indicano che lo stato della richiesta è andato a buon fine; in particolare, 200 (OK) rappresenta una chiamata che è andata a buon fine, 201 (Created) è associato alla creazione di un nuovo elemento, mentre 204 (NoContent) viene utilizzato quando la risposta non ha un contenuto vero e proprio ma è stata completata con successo;
- `300`: sono utilizzati principalmente per indicare un redirect. Pertanto nelle API possiamo utilizzarli per rimandare il client al nuovo indirizzo. Con 301 (Moved Permanently) indichiamo che la risorsa ha un nuovo URL definitivo, mentre con 302 (Found) che la stessa è disponibile temporaneamente su un altro URL;

- `400`: indicano errori avvenuti nella chiamata. Per esempio, con 401 (Unauthorized) segnaliamo che la richiesta non ha i permessi necessari a essere eseguita, mentre con 404 (Not Found) si indica una risorsa non trovata e con 405 (Method Not Allowed) che l'URL è stato richiamato con un verb HTTP non supportato;

- `500`: è una tipologia di errore lato server, in cui il client non può fare nulla, se non riprovare la chiamata successivamente.

Quelli elencati in precedenza sono solamente alcuni dei tanti codici che si possono trovare sviluppando le API o servizi web in genere, mentre altri li affronteremo nel corso di questo stesso capitolo. Pertanto, il codice dell'Esempio 11.7 può essere modificato per gestire l'esistenza di un cliente, così da gestire correttamente la risposta, come viene dimostrato nell'Esempio 11.8.

Esempio 11.8

```
[HttpGet("{id}")]
public IActionResult GetCustomer(Guid id)
{
   var customer = customerRepository.GetCustomer(id);
   if (customer == null)
      return NotFound();
   var customerForRead = Mapper.Map<CustomerForRead>(customer);
   return Ok(customerForRead);
}
```

Nell'Esempio 11.8 possiamo notare come ci siano diverse modifiche rispetto all'esempio mostrato in precedenza: la prima fra tutte è il tipo di ritorno, che non è più un `JsonResult` ma è stato reso generico sul tipo `IActionResult`: discuteremo dei vantaggi più avanti, nel corso di questo stesso capitolo.

Inoltre, abbiamo aggiunto un controllo sull'esistenza del nostro cliente: in caso in cui l'oggetto contenga il valore `null`, ovvero che non sia presente all'interno dello storage, viene rimandata la risposta a una funzione `NotFound`, che cambierà lo status code al suo corrispondente, ovvero 404 (Not Found), esattamente come ci si aspetta, senza ritornare alcun elemento o alcun valore `null`, come possiamo notare nella Figura 11.6.

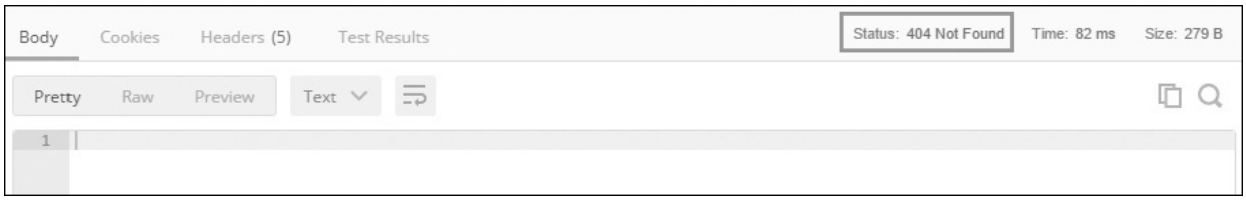


Figura 11.6 – Ecco come appare una risposta con status code 404 (Not Found), in caso di dati mancanti.

Nel caso in cui l'oggetto fosse presente, verrà restituito tramite la funzione `Ok`, che produrrà in risposta lo status code 200 (OK), come abbiamo già visto in precedenza. È interessante notare che queste due funzioni, così come altre già presenti in `Controller`, non fanno altro che wrappare una chiamata base al metodo `StatusCode`, in cui viene passato l'effettivo valore di stato. Un esempio simile potrebbe essere quello realizzato in caso di eliminazione di un elemento, in cui i valori di risposta possono differire, come viene mostrato nell'Esempio 11.9.

Esempio 11.9

```
[HttpDelete("{id}")]
public IActionResult DeleteCustomer(Guid id)
{
   if (!customerRepository.CustomerExists(id))
      return StatusCode(StatusCodes.Status404NotFound);
   customerRepository.DeleteCustomer(id);
   return StatusCode(StatusCodes.Status204NoContent);
}
```

Sebbene con questo approccio possiamo gestire i codici di stato per eventuali problemi nella chiamata effettuata lato client, non siamo ancora in grado di sfruttare a modo il codice 500 (Internal Server Error), che indica un errore del server. Questo errore, in genere, può capitare nel caso in cui ci sia un'eccezione non gestita: il client non può fare altro che tentare nuovamente la chiamata, nella speranza che il problema sia stato solo temporaneo. Una semplice chiamata che genera un'eccezione potrebbe essere quella esposta nell'Esempio 11.10.

Esempio 11.10

```
[HttpGet("api/exception")]
public IActionResult GenerateException()
{
```

```
    try
    {
       throw new NotImplementedException("Questo metodo è finto.");
    }
    catch (Exception)
    {
       return StatusCode(StatusCodes.Status500InternalServerError, "Qualcosa è andato storto.");
    }
}
```

L'eccezione è stata gestita tramite un blocco `try…catch` e, poiché siamo in grado di capire cos'è andato storto, è meglio ritornare al client l'informazione generica di errore, senza esporre troppo le informazioni relative al problema, principalmente per motivi di sicurezza. Il risultato della chiamata sarà come quello mostrato nella Figura 11.7.

Figura 11.7 – Ecco come appare un errore 500 (Internal Server Error) in presenza di eccezioni gestite.

Purtroppo, il codice che viene scritto non è detto che venga testato e sia funzionante nella sua totalità, per questo possono esistere casi in cui qualche chiamata potrebbe ancora non essere gestita e generare errori di vario tipo. In questo caso, è bene fare uso del middleware `UseExceptionHandler` nel metodo `Configure` della classe `Startup`, per modificare l'eventuale risposta prima di rimandarla al client, come possiamo notare nell'Esempio 11.11.

Esempio 11.11

```
app.UseExceptionHandler(options =>
{
   options.Run(async context =>
   {
      context.Response.StatusCode = StatusCodes.Status500InternalServerError;
      await context.Response.WriteAsync("Qualcosa è andato storto.");
   });
});
```

A tutti gli effetti, il blocco `try…catch` dell'Esempio 11.10 diventa superfluo: tutte le richieste, a questo punto, verranno filtrate e gestite dal middleware, prima di arrivare al client, ottenendo, di fatto, lo stesso risultato di mettere un blocco `try…catch` direttamente dentro ciascuna action. Le possibili eccezioni non è detto che si verifichino solamente nel codice: infatti, è anche possibile che si verifichino problemi durante l'invio della chiamata dal client verso il server, soprattutto quando i due cercano di parlare "lingue" diverse. Per questo motivo, è necessario specificare a entrambi i meccanismi con cui devono comunicare.

Formatter e content negotiation

Quando si parla di WebAPI si dà spesso per scontato che il formato della risposta ottenuta dai servizi debba per forza di cose essere in JSON ma, come è già stato illustrato all'inizio di questo capitolo, JSON è solo il formato più diffuso e pertanto il formato di rappresentazione dei dati è piuttosto indifferente. Inoltre, ci sono sistemi che lavorano con altri formati, per esempio XML oppure, nel caso di device IoT (Internet Of Things), con formati personalizzati e talvolta proprietari.

ASP.NET Core è in grado di supportare tutti gli scenari, perché tutta la sua pipeline è composta da micro-servizi sostituibili: qualora fosse necessario specificare il formato in ingresso o in uscita dal servizio, ASP.NET Core sarebbe in grado di consentircelo. A un livello più basso, però, è opportuno specificare in una sorta di contratto come client e server devono comunicare, e questo viene solitamente fatto tramite le header HTTP, con una tecnica che viene definita content negotiation.

Se il client che invia la richiesta specifica (tramite l'header `Accept`) un valore corrispondente ad `application/xml`, vuole assicurarsi che la risposta sia in formato XML. Il server, al contrario, può specificare quali formati è in grado di gestire e, eventualmente, rifiutare le chiamate con uno status code 406 (Not Acceptable), come illustrato nell'Esempio 11.12.

Esempio 11.12

```
public void ConfigureServices(IServiceCollection services)
{
```

```
   services.AddMvc(options =>
   {
      options.ReturnHttpNotAcceptable = true;
   });
}
```

La trasformazione tra oggetti e contenuto formattato in JSON piuttosto che in XML o in altri formati personalizzati viene effettuata dal runtime, in automatico, al momento dell'invio della risposta verso il client tramite i `Formatters`. Un esempio di registrazione è disponibile nell'Esempio 11.13.

Esempio 11.13

```
services.AddMvc(options =>
{
   options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
});
```

Nell'Esempio 11.13 è stato aggiunto il formatter per la codifica degli oggetti in formato XML, oltre a quelli di default già registrati, come JSON. Per costruirne uno personalizzato è necessario creare una classe che erediti da `TextOutputFormatter`, che implementi la logica prevista, e poi aggiungerlo alla lista degli `OutputFormatters`. Il formatter viene scelto in automatico dal framework in base al valore che è in grado di leggere dall'header di `Accept` durante l'esecuzione della richiesta ma, qualora questo non venga specificato, verrà ritornato il valore di default, ovvero il primo nell'elenco dei formatter. Supponendo di richiedere l'elenco dei clienti in formato XML, l'output ottenuto dalla richiesta sarà simile a quello della Figura 11.8.

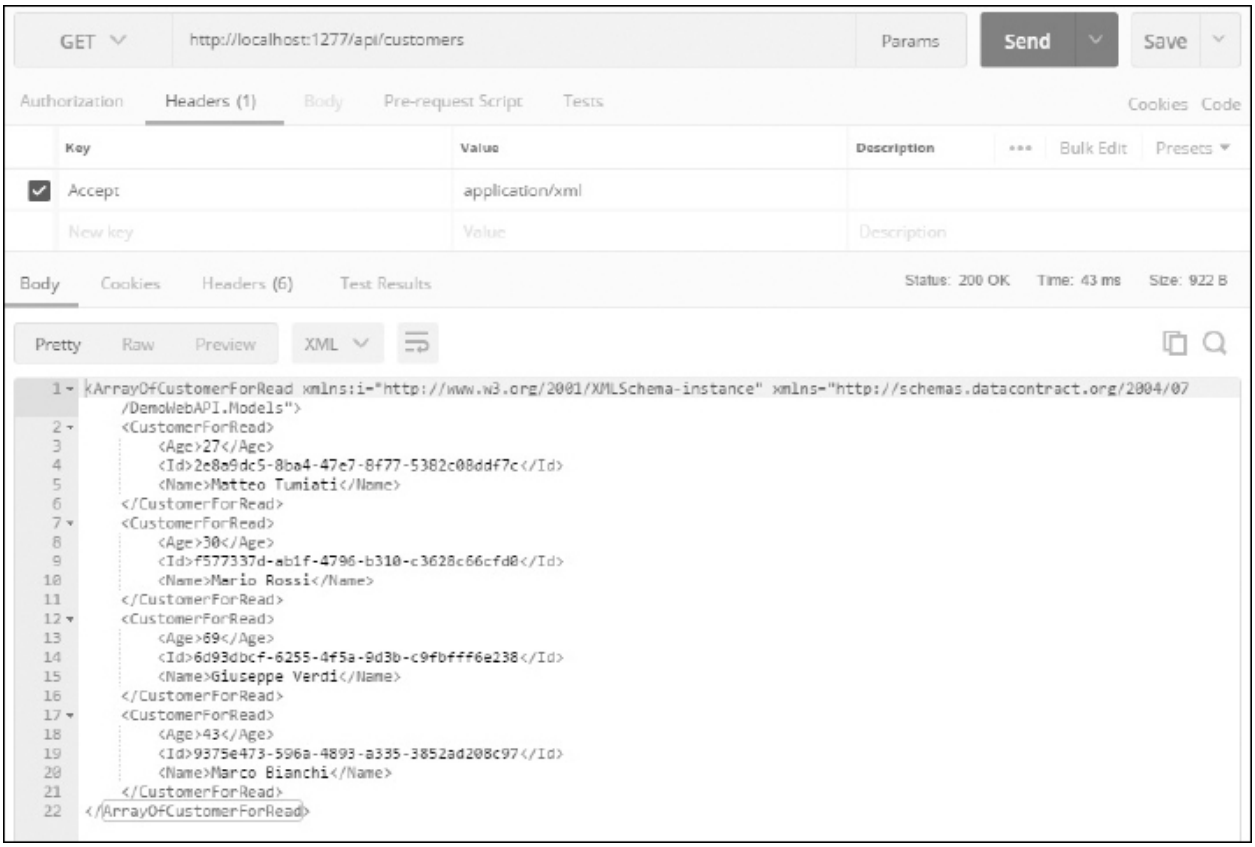


Figura 11.8 – Risposta con serializzazione in formato XML per via della richiesta con header di `Accept`.

Quello che non abbiamo ancora visto è la creazione di un'API che consenta di aggiungere un nuovo cliente. Quando si effettua la creazione, solitamente è bene utilizzare l'adeguato verb HTTP `POST` e caricare i dati che devono essere interpretati da ASP.NET Core nel body della richiesta, come è visibile nell'Esempio 11.14.

Esempio 11.14

```
[HttpPost]
public IActionResult CreateCustomer([FromBody] CustomerForCreate customer)
{
   if (customer == null || !ModelState.IsValid)
      return BadRequest();
   var mappedCustomer = Mapper.Map<Customer>(customer);
```

```
   customerRepository.AddCustomer(mappedCustomer);
   if (!customerRepository.SaveChanges())
      return StatusCode(StatusCodes.Status500InternalServerError, ”Impossibile salvare le
modifiche.”);
   var createdCustomer = Mapper.Map<CustomerForRead>(mappedCustomer);
      return CreatedAtAction(nameof(GetCustomer), new { id = createdCustomer.Id }, createdCustomer);
}
```

In questo caso dobbiamo notare l’attributo `FromBody`, che indica al model binder di recuperare e trasformare il contenuto del body della richiesta in un oggetto di tipo `CustomerForCreate`. Questa nuova classe viene utilizzata per separare completamente il DTO di lettura da quello di scrittura: in questo caso, non c’è più bisogno di proprietà come `Id`, dato che sono calcolate in fase di inserimento, oppure del nome completo, dato che i campi sul database saranno fisicamente separati in due colonne distinte. Una volta ottenuto l’oggetto ed entrati nella funzione, è necessario controllare che sia stato ricostruito correttamente e che tutte le proprietà siano valide.

Successivamente, tramite AutoMapper, andiamo a convertire l’oggetto da `CustomerForCreate` a `Customer`, in modo che possa essere persistito nella base dati. Qualora la funzione di salvataggio non andasse a buon fine, dobbiamo inviare uno status code 500 (Internal Server Error) per segnalare che l’operazione non è stata completata con successo, in modo che il client possa riprovare a eseguirla in futuro. Se, al contrario, il salvataggio si completa correttamente, è utile rimappare l’oggetto recuperato (che ora ha la proprietà `Id` configurata in modo opportuno) in un oggetto da restituire al client: così facendo, il client è in grado di capire che l’oggetto è stato effettivamente creato e lo può già utilizzare a partire dalla risposta ottenuta che, grazie alla funzione `CreatedAtAction`, ritornerà proprio uno status code di 201 (Created), come possiamo vedere nella Figura 11.9.

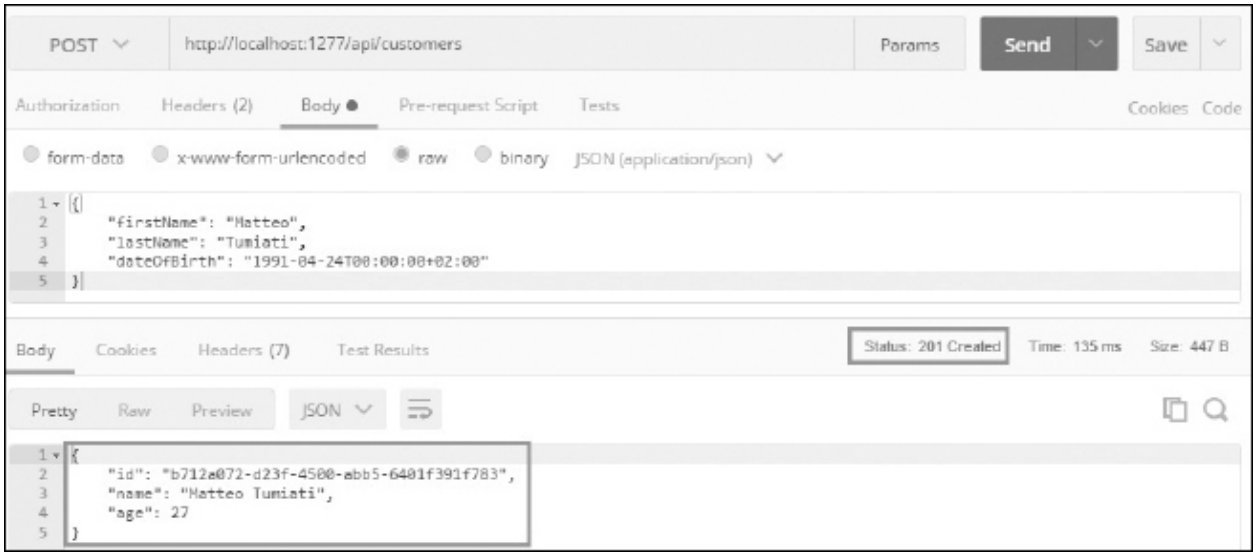


Figura 11.9 – Ecco come avviene una risposta 201 (Created) in caso di creazione di un nuovo oggetto `Customer`.

La risposta di tipo `CreatedAtAction` non ha restituito al client solo lo status code 201 (Created) e l’oggetto effettivamente creato, ma è anche andata a impostare l’header `Location` in modo che punti alla route che consente di recuperare quello stesso cliente appena creato.

Figura 11.10 – L’header `Location` in risposta a uno status code 201 (Created) serve a rimandare il client al nuovo URL, da cui recuperare le informazioni circa l’oggetto appena creato.

Tutta questa introduzione sull'inserimento di un nuovo elemento è utile per capire il concetto relativo all'header `Content-Type`: al contrario dell'header `Accept`, che è in grado di fare content negotiation sulla risposta, l'header `Content-Type` è in grado di fare la negoziazione sul formato del contenuto inviato all'interno del body della richiesta. Nella Figura 11.9 possiamo notare come questa header venga impostata automaticamente da Postman durante la scrittura del body, una volta che è stato in grado di riconoscere il testo come JSON. Questa negoziazione, però, ha senso solamente se anche lato server esiste il provider in grado di deserializzare e convertire il body nel formato corretto: nel caso di `Accept`, poiché i dati dovevano essere mandati in output, si sono utilizzati gli `OutputFormatters`, mentre nel caso di `Content-Type`, poiché i dati devono essere manipolati in fase di input, si utilizzano gli `InputFormatters`. Nell'Esempio 11.15 registriamo entrambi.

Esempio 11.15

```
services.AddMvc(options =>
{
   options.ReturnHttpNotAcceptable = true;
   options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
   options.InputFormatters.Add(new XmlDataContractSerializerInputFormatter());
});
```

Abbiamo configurato anche un convertitore dal formato XML per i dati in ingresso ma, allo stesso modo degli `OutputFormatters`, è relativamente immediato costruire un formatter personalizzato secondo le proprie esigenze e i propri protocolli. Poiché non è fra gli scopi del libro quello di entrare in scenari troppo avanzati e costruiti su misura, lasciamo al lettore un rimando alla documentazione ufficiale, necessaria per poter costruire un formatter custom. La potete trovare all'indirizzo: `http://aspit.co/bnm`.

Finora quello che è stato affrontato ha riguardato l'elaborazione di singoli elementi. Ma come bisogna comportarsi nei casi in cui ci siano grandi quantità di dati da spostare da un sistema a un altro?

Lavorare con le collection

Ci sono diversi casi, come quello del bulk insert, in cui si può verificare la necessità di caricare i dati da una collection. Per come è stato strutturato il routing, non c'è una modalità per caricare una lista di dati con una route univoca, infatti/api/customers è già stata utilizzata per lavorare con i singoli oggetti. La soluzione più immediata a questo problema è di creare una nuova risorsa, con un nuovo controller, specifico per lavorare con array di oggetti, come viene mostrato nell'Esempio 11.16.

Esempio 11.16

```
[HttpPost]
[Route("/api/customerscollection")]
public IActionResult CreateCustomerCollection([FromBody] IEnumerable<CustomerForCreate> customers)
{
   if (customers == null || !customers.Any())
      return BadRequest();
   var mappedCustomer = Mapper.Map<IEnumerable<Customer>>(customers);
   customerRepository.AddCustomers(mappedCustomer);
   if (!customerRepository.SaveChanges())
      return StatusCode(StatusCodes.Status500InternalServerError, "Impossibile salvare le
modifiche.");

   return StatusCode(StatusCodes.Status201Created);
}
```

Il codice dell'Esempio 11.16 è molto simile a quello che è già stato introdotto in precedenza per l'inserimento di un singolo elemento. Le differenze si trovano nel fatto che la action risponde sulla route `/api/customerscollection` anziché `/api/customers` e nel fatto che con l'attributo `FromBody` si va a recuperare una intera collezione di elementi, non solo uno; come sempre sarà ASP. NET Core a fare il mapping secondo i formatter e secondo il `Content-Type` specificato a livello di richiesta. L'ultima differenza importante che possiamo notare nell'esempio riportato sopra è che, nel caso in cui sia andato a buon fine l'inserimento, anziché ritornare una `CreatedAtAction` che va a creare un header `Location` nella risposta e l'elenco degli oggetti creati, si ritorna solamente lo status code 201 (Created). Questa forse non è la pratica migliore, ma è questione di scelte: supponendo di dover caricare migliaia di dati, la risposta potrebbe diventare molto grande se dovessimo ricostruire tutti gli oggetti e ritornarli e, allo stesso modo, l'URL generato nel `Location` header dovrebbe contenere tutti gli identificativi univoci delle risorse, tra l'altro facendo uso di un model binder personalizzato di cui non abbiamo ancora parlato ma che affronteremo più avanti nel corso del libro. L'elaborazione dei dati, però, non riguarda solamente la lettura o la creazione di nuovi elementi ma anche l'aggiornamento degli stessi, come vedremo a breve.

Eseguire aggiornamenti

Negli esempi illustrati in precedenza, si è sempre parlato di come affrontare un inserimento di una singola entità e di una collezione, oppure della lettura dei dati: ma come bisogna comportarsi in caso di un aggiornamento di una entità? Le strade da percorrere sono due e la scelta di una o dell'altra è dettata dalla logica di business. La prima strada possibile è quella che comporta l'aggiornamento completo della risorsa, come illustra l'Esempio 11.17, tramite l'uso del verb PUT.

Esempio 11.17

```
[HttpPut("{id}")]
public IActionResult UpdateCustomer(Guid id, [FromBody] CustomerForUpdate customer)
{
   if (customer == null)
      return BadRequest();
   var mappedCustomer = Mapper.Map<Customer>(customer);
   mappedCustomer.Id = id;
   customerRepository.UpdateCustomer(mappedCustomer);
   if (!customerRepository.SaveChanges())
      return StatusCode(StatusCodes.Status500InternalServerError, "Impossibile salvare le
modifiche");

   return NoContent();
}
```

In questo esempio, la classe `CustomerForUpdate` è una variante di quelle viste in precedenza, per mantenere un certo livello di separazione: nel caso di un aggiornamento, infatti, per una scelta di logica, potremmo decidere che sia consentita la sola modifica dei campi nome e cognome. La proprietà `Id` che va a identificare in modo univoco il cliente da aggiornare viene recuperata dalla query string tramite il matching dell'attributo `Id`, mentre il cliente arriva dopo la content negotiation dal body e, una volta rimappato con AutoMapper, viene eseguito l'aggiornamento, come mostrato nell'Esempio 11.18.

Esempio 11.18

```
public void UpdateCustomer(Customer customer)
{
   var internalCustomer = customers.FirstOrDefault(x => x.Id == customer.Id);
   internalCustomer.FirstName = customer.FirstName;
   internalCustomer.LastName = customer.LastName;
}
```

La richiesta inviata con Postman, in caso di cliente esistente si completerà correttamente, come ci si aspetta, con uno status code 204 (No Content), poiché non abbiamo specificato un tipo di ritorno per la action.

All'inivio con PUT di un oggetto di tipo `CustomerForUpdate` valido ma con la proprietà `LastName` non impostata, si otterrà una risposta con successo perché, a tutti gli effetti, non si sono verificati errori di alcun tipo e la validazione è stata superata senza problemi. Poiché viene rifatto l'assegnamento nell'`UpdateCustomer`, si otterrà un valore `null` anziché il valore precedente invariato.

Sebbene questo comportamento possa essere corretto in scenari come quello dell'esempio, in cui sono presenti solo due campi, in caso di logiche più complesse in cui si trattino dati con decine di proprietà, fare uso della PUT e reinviare ogni volta tutto il payload non è la soluzione migliore, anche a livello di banda. Per questo esiste la possibilità di fare aggiornamenti parziali tramite il verb `PATCH`.

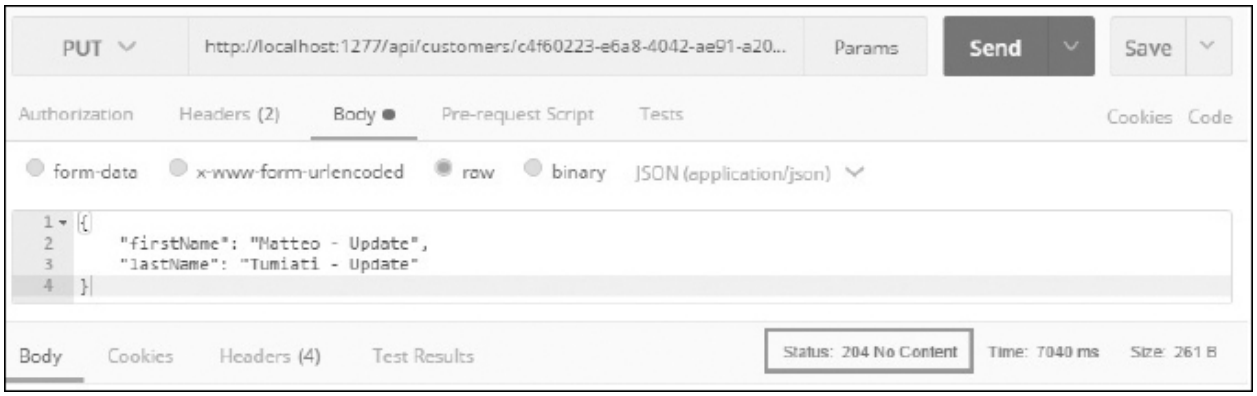


Figura 11.11 – L'aggiornamento completo dei dati di un cliente è possibile con una chiamata con verb PUT che restituirà come risultato 204 (No Content) in caso di update andato a buon fine.

L'operazione di PATCH, all'opposto della PUT, non accetta più in ingresso tutto l'oggetto `CustomerForUpdate` ma un oggetto di tipo `JsonPatchDocument`, che richiede anche un nuovo media-type `application/json-patch+json` per essere gestito. Si tratta di un file JSON, che include un elenco di operazioni che possiamo effettuare sugli oggetti inviati:

- `Add`: consente l'aggiunta di una nuova proprietà con il valore specificato, in casi in cui si stia lavorando con proprietà di tipo dynamic.
- `Remove`: invalida il contenuto della proprietà selezionata e lo imposta al suo valore di default (per esempio `null`).
- `Replace`: aggiorna il valore della proprietà selezionata con quella specificata nel documento.
- `Copy`: copia il valore di una proprietà verso un'altra.
- `Move`: è come l'operazione di `Copy` ma, in più, effettua anche una `Remove` sul percorso specificato.
- `Test`: verifica che il valore contenuto nel percorso richiesto sia identico a quello specificato nella richiesta.

Un JSON di un documento di questo tipo è disponibile nell'Esempio 11.19.

Esempio 11.19

```
[
      {
         "op": "replace",
         "path": "/firstName",
         "value": "Matteo (Updated)"
      }
]
```

L'Esempio 11.19 mostra come deve essere costruito il documento che deve essere inviato. In particolare, possiamo notare come venga richiesta, tramite la proprietà `op`, la sostituzione del valore contenuto in `FirstName` con il valore contenuto nella proprietà `value`. Poiché è a tutti gli effetti un array, si possono concatenare diverse operazioni da effettuare in sequenza ma, per semplicità e per capirne il funzionamento, è più che sufficiente elaborare un solo parametro. L'Esempio 11.20 mostra un nuovo metodo capace di accettare un documento di questo tipo.

Esempio 11.20

```
[HttpPatch("{id}")]
public IActionResult PartialUpdateCustomer(Guid id, [FromBody] JsonPatchDocumen t<CustomerForUpdate>
patchDocument)
{
   if (patchDocument == null)
      return BadRequest();
   var customer = customerRepository.GetCustomer(id);
   var mappedCustomer = Mapper.Map<CustomerForUpdate>(customer);
   patchDocument.ApplyTo(mappedCustomer);
   customer = Mapper.Map<Customer>(mappedCustomer);
   customer.Id = id;
   customerRepository.UpdateCustomer(customer);
   if (!customerRepository.SaveChanges())
      return StatusCode(StatusCodes.Status500InternalServerError, "Impossibile salvare le
modifiche");

   return NoContent();
}
```

A livello di logica il flusso d'esecuzione di una chiamata PATCH è simile a quello già visto per una chiamata con PUT, ma ci sono un paio di differenze: la prima è che a livello di body non arriva più l'oggetto `CustomerForUpdate` ma un `JsonPatchDocument<CustomerForUpdate>`. La seconda differenza è che prima di richiamare l'aggiornamento sul repository viene applicata la modifica sul `JsonPatchDocument`, pertanto tutte le proprietà specificate nel documento verranno aggiornate, eliminate o aggiunte, mentre le altre rimarranno invariate, andando di fatto a risolvere il problema posto dalla PUT.

Con la gestione degli aggiornamenti va a concludersi l'overview relativa alle operazioni sui dati che sono necessarie per realizzare un sistema basato su WebAPI ma, come abbiamo già detto più volte, realizzare un'API non è la stessa cosa di scrivere un servizio RESTful. Pertanto ora vediamo come integrare le parti mancanti per avere un servizio completo.

Hypermedia As The Engine Of Application State (HATEOAS)

È già stato anticipato all'inizio del capitolo: lavorare e costruire delle WebAPI funzionali e funzionanti non è detto che coincida con lo sviluppo di un vero e proprio servizio RESTful. Il pezzo mancante alle WebAPI costruite per diventare un servizio REST a tutti gli effetti è la parte descrittiva perché, per quanto la parte server sia ben strutturata, è difficile pensare di avere un client che agisca ed evolva in modo completamente indipendente dalla parte server. Quali operazioni possiamo fare su una determinata risorsa? Possiamo aggiornare il dato in modo completo o anche in modo parziale? È possibile eliminare una risorsa? È possibile avere la paginazione nel caso in cui ci siano troppi dati da leggere? Queste sono solo alcune delle domande alle quali bisogna dare una risposta con un sistema auto-descrittivo perché, al momento, la logica presente nelle demo precedenti va solamente a lavorare con i dati e non con i metadata.

HATEOAS (Hypermedia As The Engine Of The Application State) è una serie di principi già ampiamente utilizzati in ambiente web e che consente di aggiungere in risposta una serie di link, o di hypermedia, come se fossero dei metadati: in HTML, l'equivalente sarebbe l'anchor element, che include tramite `href` l'indirizzo sul quale recuperare la risorsa, in `rel` come si relaziona alla risorsa stessa e infine `type`, opzionale, che include il media-type. Tradotto in codice, quello che serve è una classe che gestisca queste tre nuove proprietà, come viene mostrato nell'Esempio 11.21.

Esempio 11.21

```
public class Link
{
   public string Href { get; set; }
   public string Rel { get; set; }
   public string Method { get; set; }
}
```

Nel caso delle WebAPI, il media-type viene gestito in un modo alternativo, come abbiamo già visto in precedenza in questo stesso capitolo, per questo è necessario interessarsi maggiormente su quale verb (o metodo) HTTP può essere invocato su un determinato URL. Poiché sono gli oggetti ritornati a dover includere i link, è necessario includere la proprietà corrispondente nel modello oppure estendere l'esistente. L'Esempio 11.22 mostra come fare.

Esempio 11.22

```
public class CustomerForReadExtended : CustomerForRead
{
   public ICollection<Link> Links { get; set; } = new List<Link>();
}
```

A questo punto, bisogna trovare un metodo per ricostruire il link corretto a partire da una action, ma per fortuna questo lavoro è già implementato dalla classe `UrlHelper` di ASP.NET Core, che va registrata appositamente come servizio aggiuntivo allo startup, come viene mostrato nell'Esempio 11.23.

Esempio 11.23

```
public void ConfigureServices(IServiceCollection services)
{
   // ...
   services.AddSingleton<IActionContextAccessor, ActionContextAccessor>();
   services.AddScoped<IUrlHelper>(factory =>
   {
      var actionContext = factory.GetService<IActionContextAccessor>()
                                                  .ActionContext;
      return new UrlHelper(actionContext);
  });
}
```

L'interfaccia `IUrlHelper` della classe corrispondente può quindi essere iniettata nel costruttore del `CustomerController` e utilizzata per aggiungere i link corretti all'oggetto `Customer`, così da recuperare correttamente gli URL, come mostrato nell'Esempio 11.24.

Esempio 11.24

```
private CustomerForReadExtended AddLinks(CustomerForReadExtended customer)
{
   customer.Links.Add(new Link(urlHelper.Link("GetCustomer", new { id = customer.Id }), "self",
"GET"));
   customer.Links.Add(new Link(urlHelper.Link("CreateCustomer", null), "self", "POST"));
   customer.Links.Add(new Link(urlHelper.Link("UpdateCustomer", new { id = customer.Id }),
                                                                    "update_customer",
                                                                    "PUT"));
```

```
   customer.Links.Add(new Link(urlHelper.Link("PartialUpdateCustomer", new { id = customer.Id }),
                                                                    "partial_update_customer",
                                                                    "PATCH"));
   return customer;
}
```

Come possiamo notare nell'Esempio 11.24, vengono aggiunti i link nella lista prevista dall'oggetto `Customer` e vengono costruiti partendo dalle proprietà `Name` aggiunte sopra le corrispettive action, come viene illustrato nell'Esempio 11.25.

Esempio 11.25

```
[HttpGet("{id}", Name = "GetCustomer")] public IActionResult GetCustomer(Guid id)
{
}
```

Per costruire correttamente gli URL, è necessario passare anche eventuali parametri, come per esempio gli identificativi dei clienti, quindi andiamo ad aggiungere i valori relativi a `Rel`, e `Method` (o `Type`): `Rel` è rappresentato dal valore `SELF` nei casi in cui la chiamata venga fatta sulla stessa URL del chiamante, mentre ha un altro valore altrettanto descrittivo quando si riferisce ad action differenti. La proprietà `Method` è valorizzata con il corrispettivo verb HTTP che la chiamata si aspetta (per esempio, `POST` in caso ci si riferisca all'URL di creazione di un nuovo oggetto). L'ultimo passaggio da eseguire è quello di cambiare la risposta per la action `GetCustomer`, come viene evidenziato nell'Esempio 11.26.

Esempio 11.26

```
[HttpGet("{id}", Name = "GetCustomer")]
public IActionResult GetCustomer(Guid id)
{
   var customer = customerRepository.GetCustomer(id);
   if (customer == null)
      return NotFound();
   var customerForRead = Mapper.Map<CustomerForReadExtended>(customer);
   return StatusCode(StatusCodes.Status200OK, AddLinks(customerForRead));
}
```

Il risultato ottenuto è visibile infine nella Figura 11.12, che illustra perfettamente come siano visibili tutti i link necessari per realizzare altre richieste al server sulla stessa risorsa di tipo cliente.

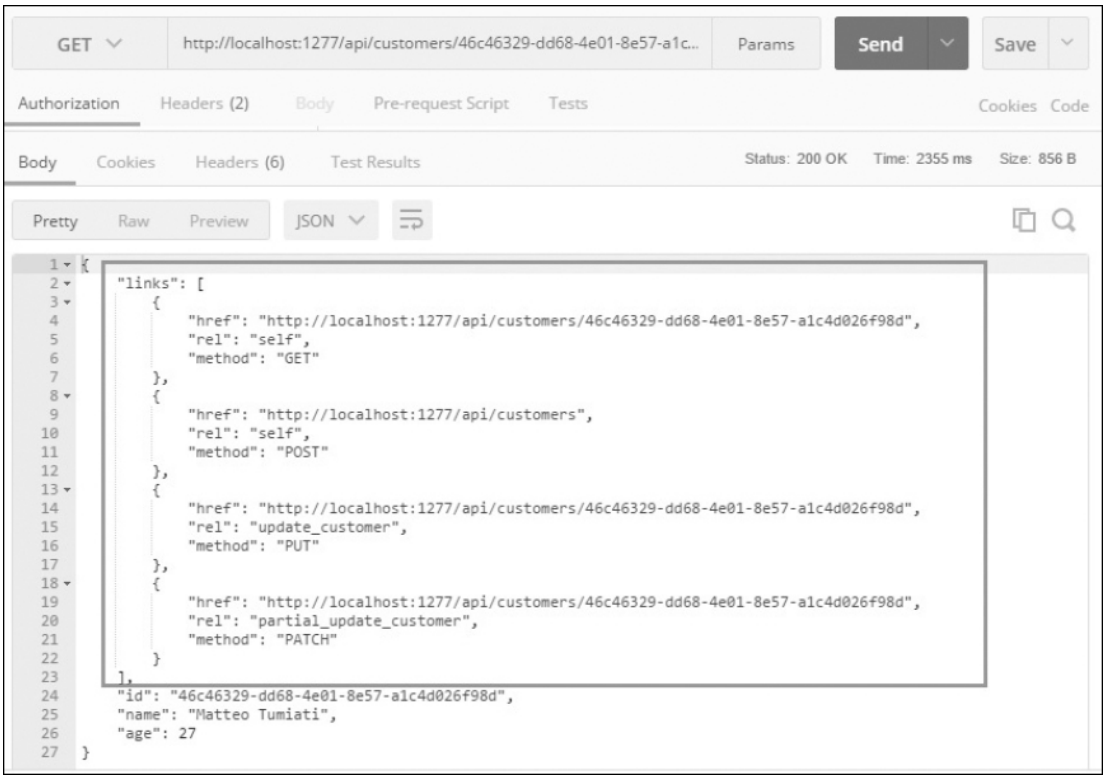


Figura 11.12 – I link in risposta a una chiamata in `GET` fungono da metadati per il self-discovery lato client e vengono inviati come parte del payload di risposta.

Nonostante questa tecnica sia piuttosto diffusa nel mondo delle REST API, implica la creazione di una funzione personalizzata per ogni metodo. In alternativa o, meglio, in aggiunta a quanto già realizzato con i link, ci sono ancora due verb HTTP che possiamo sfruttare per dare una maggior indicazione del servizio, in modo tale che si possa fare auto-discovery degli endpoint: si tratta di OPTIONS e HEAD.

Una richiesta inviata come OPTIONS è utile perché il client può fare il discovery dei metodi che sono abilitati a rispondere su un determinato URL, senza però ricevere in risposta l'effettivo risultato. In ASP.NET Core WebAPI viene gestita come è evidenziato nell'Esempio 11.27.

Esempio 11.27

```
[HttpOptions]
public IActionResult GetCustomersOptions()
{
   Response.Headers.Add("Allow", "GET,POST,OPTIONS");
   return Ok();
}
```

La chiamata all'endpoint /api/customers con il verb OPTIONS non fa altro che aggiungere un nuovo header, chiamato Allow, in cui sono elencati in forma di stringa e separati da virgola tutti i verb HTTP che rispondono sullo stesso endpoint. In questo caso specifico, come possiamo notare, non vengono elencati DELETE, PATCH e PUT poiché questi in realtà rispondono su un URL differente, che include anche l'identificativo del cliente, ed è proprio per questo motivo che questo metodo andrebbe usato assieme a quello dei link visti in precedenza. In qualsiasi caso, anche in quelli in cui non ci sono verb HTTP che supportano l'endpoint richiesto, la chiamata deve rispondere con uno status code di 200 (OK) se la action è marcata con l'attributo HttpOptions. Il risultato di una chiamata è evidenziato nella Figura 11.13.

Figura 11.13 – L'header Allow mostra l'elenco dei metodi HTTP che sono utilizzabili sull'URL appena chiamato.

La chiamata con verb HEAD invece, può essere fatta su tutte le action che dichiarano anche altri verb, come GET per esempio, e viene utilizzata principalmente per testare se i parametri passati in ingresso (query string, header e body) e le header in uscita sono validi per poter considerare l'integrazione dell'API lato client come funzionante: non verrà ritornato, infatti, alcun contenuto in risposta, anche per le chiamate in GET, proprio perché non è necessario e sarà direttamente il runtime a gestire l'output, senza che ci sia la necessità di farlo all'interno della action. L'unica modifica è quella di marcare la funzione corrispondente con l'apposito attributo, come viene mostrato nell'Esempio 11.28.

Esempio 11.28

```
[HttpGet]
[HttpHead]
public IActionResult GetCustomers()
{
   // ...
}
```

La stessa API che risponde all'URL /api/customers supporterà chiamate sia in GET sia in HEAD, come viene evidenziato nella Figura 11.14, ma non produrrà alcun output.

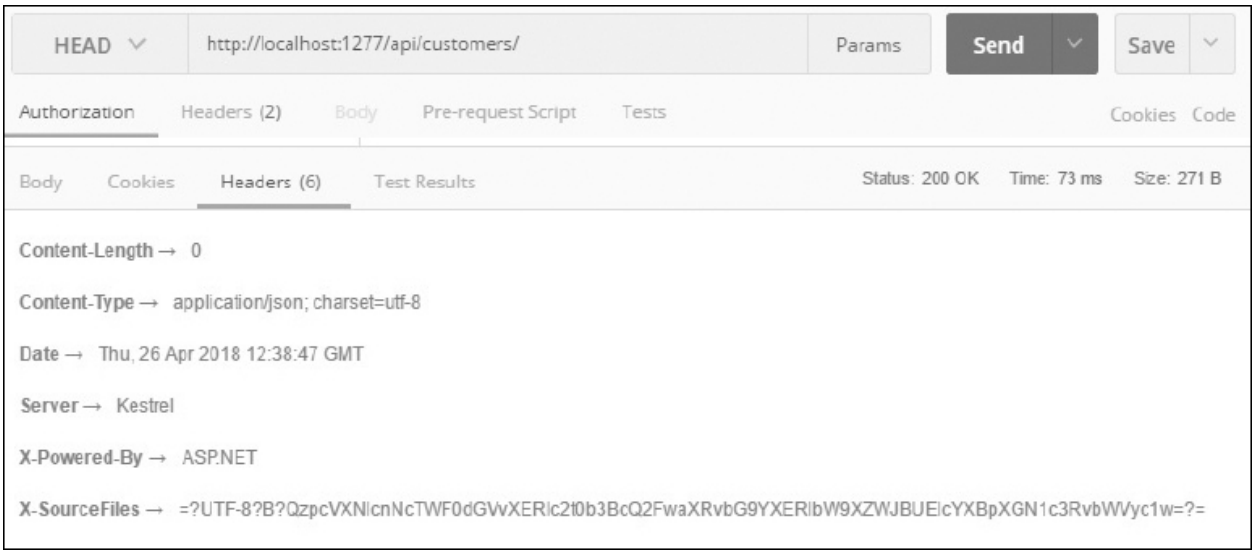


Figura 11.14 – Una chiamata con verb HEAD non restituisce alcun risultato ma mantiene tutti gli header e gli status code in risposta.

Nonostante non venga riportato il body nella risposta, sarà comunque possibile, grazie al servizio di self-discovery e in aggiunta alla chiamata fatta in precedenza con OPTIONS, avere un sistema di WebAPI completamente autonomo, testabile e riproducibile.

Conclusioni

In questo capitolo sono stati trattati due temi di eguale importanza che riguardano il tema della comunicazione: il primo, forse il più utilizzato secondo le esigenze odierne, è il mondo delle WebAPI, che permette di esporre come servizio su un preciso endpoint i dati che vengono prelevati direttamente dal database o da una qualsiasi altra fonte di dati attraverso uno o più strati applicativi in modo proporzionale alla complessità del proprio sistema, anche se per semplicità abbiamo solamente affrontato il tema del repository pattern. Successivamente, abbiamo parlato delle differenze che ci sono nel lavorare con le WebAPI, standard di ASP.NET Core, e quali sono i vantaggi nel trasformare queste ultime, facendo anche uso di HATEOAS in RESTful API, per avere maggiore separazione tra client e server, pur mantenendo quelli che sono i principi base di funzionamento. Una volta preparato il design e l'architettura delle API, è necessario pensare a come effettuarne il mantenimento e il versionamento.

Nel prossimo capitolo vedremo degli scenari avanzati e discuteremo non solo quali strategie e quali strumenti possano essere utilizzati per quest'ultimi due argomenti, ma vedremo anche i dettagli di un nuovo sistema di comunicazione, i WebHook, e parleremo delle novità riguardanti una infrastruttura per comunicazione in tempo reale.

12
Sviluppare servizi RESTful avanzati con ASP.NET Core WebAPI, WebHook e SignalR

Nel corso del capitolo precedente abbiamo introdotto quelli che sono i concetti relativi alle WebAPI e, in particolare, abbiamo parlato di come ASP.NET si sia evoluto per integrare nel framework di MVC anche tutta la parte dei `WebApiController`. Abbiamo inoltre discusso di quali sono le differenze tra lo sviluppo di una WebAPI standard, comunque accettabile e funzionale, e creare un vero e proprio servizio RESTful, introducendo concetti relativi all'uso dei metadati per supportare HATEOAS e avere il massimo dell'indipendenza, per permettere sviluppi concorrenti tra client e server.

Sempre in quest'ottica nascono i WebHook, ovvero delle callback HTTP che permettono l'estensione di una WebAPI con un'altra, invocata in modo asincrono e che potrebbe appartenere a vendor differenti, consentendo, di fatto, l'estensione del servizio anche lato server. Ci sono invece casi, all'opposto di questi, in cui il client ha la necessità di comunicare senza latenza con il server, come per esempio servizi esposti dal mondo della finanza oppure, tipicamente, i giochi. Nonostante lo scenario sembri avanzato e complesso, nel corso del capitolo parleremo di come uno strumento come SignalR, integrato in ASP.NET Core, venga incontro alle esigenze di tutti gli sviluppatori per semplificare gli scenari di comunicazione real-time.

Una volta costruito tutto il sistema di comunicazione tramite API, WebHook e SignalR, così come per tutto quello che riguarda il mondo del software, bisogna pensare alla manutenzione: è fondamentale prevedere il servizio come aggiornabile per quei casi in cui ci sia un cambio del modello dei dati, oppure nei meccanismi di elaborazione, pertanto è necessario lavorare con un sistema che supporti il versionamento. Infine, imparare a scrivere la corretta documentazione ed esporla a sua volta come se fosse un servizio, in costante aggiornamento, è d'obbligo per farsi immediatamente un'idea del contesto e funzionamento delle API stesse.

Documentazione con Swagger

La scrittura delle API è completata ma, come abbiamo anticipato e da buone abitudini da sviluppatori, è bene iniziare a pensare alla documentazione per lasciare traccia di quello che è stato fatto e del funzionamento per i vari client che andranno a utilizzare questi servizi. Scrivere la documentazione è un processo tedioso che difficilmente rende felici gli sviluppatori ma, per fortuna, ci sono diversi strumenti che vengono in aiuto per cercare di automatizzare il più possibile l'intero processo. Tra questi strumenti ci sono Swagger, una specifica utilizzata per documentare le API che può essere scritta in formato JSON o YAML, e Swashbuckle, uno strumento che permette di creare la documentazione per Swagger in modo automatico a partire dai commenti XML inseriti nel codice. Maggiori informazioni e la relativa documentazione su Swagger sono disponibili su `http://aspit.co/bno`.

Questi due strumenti sono già integrati in un pacchetto di NuGet da aggiungere alla soluzione, chiamato `Swashbuckle.AspNetCore`. Il passaggio successivo, come spesso accade, è la registrazione del servizio all'interno della `ConfigureServices`, come è visibile nell'Esempio 12.1.

Esempio 12.1

```
services.AddSwaggerGen(c =>
{
   c.SwaggerDoc("v1", new Info
   {
      Version = "v1",
      Title = "Documentazione WebAPI ASP.NET Core",
      Description = "Esempio di come si fa la documentazione con ASP.NET Core",
      TermsOfService = "None",
      Contact = new Contact
      {
         Name = "ASPItalia.com",
         Email = string.Empty,
         Url = "https://aspitalia.com"
      }
   });
   var xmlFile = $"{Assembly.GetEntryAssembly().GetName().Name}.xml";
   var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
   c.IncludeXmlComments(xmlPath);
});
```

I parametri definiti all'interno di questa funzione possono essere personalizzati secondo le nostre esigenze. Escludendo la configurazione di base, possiamo notare come venga anche passato un file XML recuperato dalla directory di output (o di publish): questo file contiene tutti i commenti XML inseriti sui vari metodi nel codice e deve essere abilitato in modo esplicito nella pagina Build delle proprietà del progetto, come è illustrato nella Figura 12.1.

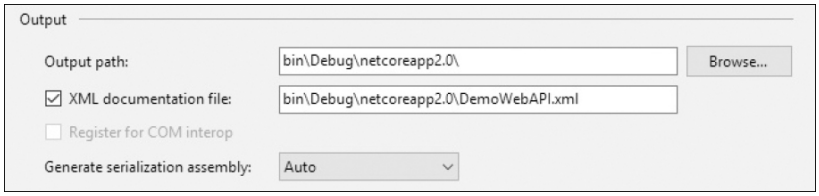


Figura 12.1 – Dalle proprietà del progetto, all'interno di Visual Studio, è possibile abilitare la pubblicazione della documentazione in formato XML nella cartella di output della build o in un qualsiasi altro percorso.

La parte più interessante arriva quando si vuole avviare il servizio: nei paragrafi precedenti è stato anticipato come Swagger funzioni su una specifica JSON o YAML, che non sono facilmente leggibili nel caso in cui ci siano molte API, ma per fortuna Swashbuckle contiene, fra le altre funzionalità, anche un motore che consente di generare dell'interfaccia grafica a partire dal file di Swagger, in modo molto intelligente e intuitivo, garantendo anche interattività con le varie API. Per abilitare l'interfaccia è sufficiente registrare Swagger e SwaggerUI nel metodo `Configure`. L'Esempio 12.2 mostra una tipica registrazione in tal senso.

Esempio 12.2

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   // ...
   app.UseStaticFiles();
   app.UseSwagger();
   app.UseSwaggerUI(c =>
   {
      c.SwaggerEndpoint("/swagger/v1/swagger.json", "Versione 1.0");
   });
   app.UseMvc ();
}
```

Come possiamo notare, nell'Esempio 12.2, è anche necessario abilitare l'uso dei file statici, altrimenti non sarà possibile leggere il file `swagger.json`. Avviando l'applicazione e navigando alla route `/swagger`, è già possibile vedere l'interfaccia grafica, come è mostrato nella Figura 12.2.

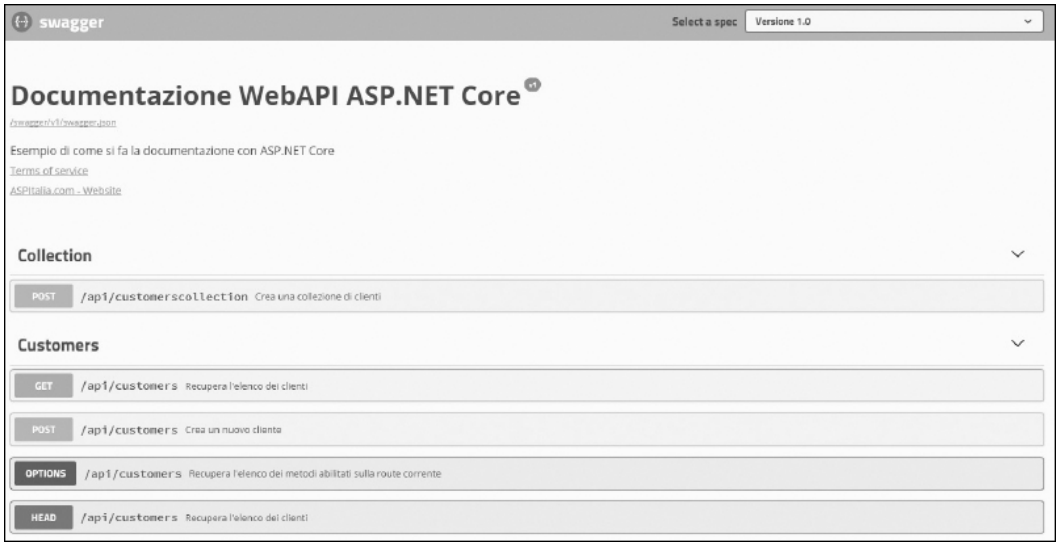


Figura 12.2 – Navigando all'URL localhost/swagger è possibile vedere l'interfaccia grafica esposta da Swagger, costruita sopra il file JSON posizionato in localhost/swagger/v1/swagger.json, oppure esposto nel percorso specificato in `SwaggerEndpoint`.

Ogni singolo elemento esposto in questa interfaccia è stato recuperato dal file XML generato in automatico dai commenti: pertanto, non solo saranno visibili le route ma sarà anche possibile vedere i tipi di ritorno, gli errori generati e disporre di una dashboard per provare le API.

La action in cui recuperiamo un cliente esistente è stata pertanto modificata dal capitolo precedente con i commenti illustrati nell'Esempio 12.3.

Esempio 12.3

```
/// <summary>
/// Recupera un cliente
/// </summary>
```

```
/// <param name="id">L'identificativo di un cliente</param>
/// <returns>Un cliente selezionato tramite il suo identificativo</returns>
/// <response code="200">Ritorna il cliente scelto tramite il suo id</response> /// <response
code="404">Se il cliente non è stato trovato</response>
[HttpGet("{id}", Name = "GetCustomer")]
[Produces("application/json", "application/xml")]
[ProducesResponseType(typeof(CustomerForRead), 200)]
[ProducesResponseType(404)]
public IActionResult GetCustomer(Guid id)
{
   var customer = customerRepository.GetCustomer(id);
   if (customer == null)
      return NotFound();
   var customerForRead = Mapper.Map<CustomerForRead>(customer);
   return StatusCode(StatusCodes.Status200OK, customerForRead);
}
```

Oltre ai commenti più classici come il `summary`, il `return` ed eventuali parametri, sono comparsi i commenti di `response code`: vengono utilizzati da Swagger per mostrare i potenziali tipi di risposta. La Figura 12.3 mostra un esempio di risposta prodotta a partire dal codice precedente.

Figura 12.3 – La combinazione dei commenti XML nel tag `response` e degli attributi `ProducesResponseType` viene letta da Swagger per mostrare in maniera visiva e naturale le possibili risposte dell'API.

A livello di attributi, invece, possiamo notare `Produces`, che indica quale media-type ci possiamo aspettare in ingresso e in risposta per la content negotiation, e `ProducesResponseType`, che specifica la tipologia di modello, ovvero la classe che ci si deve aspettare nel caso in cui ci sia una risposta con un determinato status code, solitamente 200 (OK).

Figura 12.4 – Le classi C# identificate nei parametri delle action vengono esposte da Swagger per essere utilizzate direttamente tramite l'interfaccia grafica e fare un controllo d'integrità dei tipi prima dell'invio della richiesta al server.

Nel caso in cui si voglia ottenere un'interfaccia grafica più coerente con il resto del sito web costruito, è anche possibile andare a modificare i CSS e crearne di personalizzati: è necessario creare la cartella swagger/ui all'interno di wwwroot e aggiungere i nuovi fogli di stile. Una volta organizzata ed eventualmente personalizzata la documentazione iniziale, non sarà complicato averla sempre aggiornata nel tempo, poiché richiede, tipicamente, la sola scrittura di poche righe di commento per ogni action che viene costruita. L'aggiornamento stesso della documentazione non è nemmeno detto che sia così frequente ma dipende principalmente da quanto le API stesse vengono modificate nel tempo. Per questo è utile iniziare a parlare di come vengono mantenute le versioni.

Gestione delle versioni

Nel corso degli anni un'API può evolvere, così come il codice dell'intera applicazione. I servizi REST, però, di fatto stabiliscono un contratto (anche se poco formale) e non andrebbe mai rimosso un endpoint, o, peggio ancora, variato il comportamento.

Possiamo segnalare questi casi attraverso il versioning, applicando una sorta di tag alle API per specificare che l'implementazione rimarrà identica, fino al momento in cui verrà resa deprecata ed eventualmente dismessa, ma le modifiche al modello o agli strati applicativi (per esempio quelli per recuperare i dati dal database) possono ancora esserci: per questi verrà applicato un tag diverso o, per essere più precisi, una versione maggiorata, che marcherà i servizi su nuovi endpoint.

ASP.NET Core di default non ha qualcosa di già implementato ma ci sono ben tre applicazioni che possiamo integrare:

- Versioning tramite parametri della query string: sfruttando un attributo personalizzato, per esempio `api-version`, possiamo effettuare una richiesta a un determinato URL con una specifica versione.

- Versioning sull'header: inviando un attributo personalizzato, per esempio `X-API-Version`, che può essere utilizzato per tenere traccia della versione corrente dell'API.

- Versioning sull'URL: è il link stesso a contenere una route o un host differente, per indicare il numero di versione utilizzato.

Poiché per realizzare la seconda soluzione è necessario avere un'approfondita conoscenza dei middleware, che affronteremo nei capitoli successivi, e considerando che la terza soluzione è troppo complessa, per cui la vedremo solo in parte, per il momento adotteremo la prima soluzione che, seppure non è la più elegante, è comunque funzionale.

La prima cosa da fare è aggiungere da NuGet il pacchetto `Microsoft.AspNetCore.Mvc.Versioning` che va a integrare tutte le classi necessarie per gestire il versionamento, per poi registrare il servizio al momento dello startup nella `ConfigureServices`, come mostrato nell'Esempio 12.4.

Esempio 12.4

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddApiVersioning(config =>
   {
      config.AssumeDefaultVersionWhenUnspecified = true;
      config.DefaultApiVersion = new ApiVersion(1, 0);
   });
}
```

Durante la fase di configurazione è stato necessario specificare che la versione di default utilizzata è la "1.0", questo anche per via del fatto che è l'unica attualmente supportata. Inoltre, è stata anche impostata su `true` la proprietà `AssumeDefaultVersionWhenUnspecified` che permette di fare il fallback in automatico alla versione di default qualora non dovesse essere specificata nella query string. Per testarne il funzionamento, è importante impostare i due controller sulla stessa route (per fare in modo che si chiamino nello stesso nome è sufficiente cambiare il nome del namespace), come viene mostrato nell'Esempio 12.5.

Esempio 12.5

```
namespace DemoWebAPI.Controllers
{
      [ApiVersion("1.0")]
      [Route("api/versioning")]
      public class VersioningController : Controller
      {
         [HttpGet]
         public IActionResult Get()
         {
            return Ok("v1");
         }
      }
```

```
}

namespace DemoWebAPI.Controllers.V2
{
      [ApiVersion(”2.0”)]
      [Route(”api/versioning”)]
      public class VersioningController : Controller
      {
         [HttpGet]
         public IActionResult Get()
         {
            return Ok(”v2”);
         }
      }
}
```

Nell’Esempio 12.5 possiamo notare come entrambe le API abbiano sopra l’attributo `Route` un nuovo attributo, che introduciamo ora, chiamato `ApiVersion`. Al contrario di quello che avviene in fase di startup, l’attributo `ApiVersion` accetta una stringa, quindi è fondamentale prestare attenzione durante la scrittura delle versioni, altrimenti si potrebbe correre il rischio di fare sempre fallback sulla versione di default. La chiamata fatta specificando (o non) in query string il valore `api-version` farà variare la risposta da “v1” a “v2”, come viene mostrato nell’immagine 12.5.

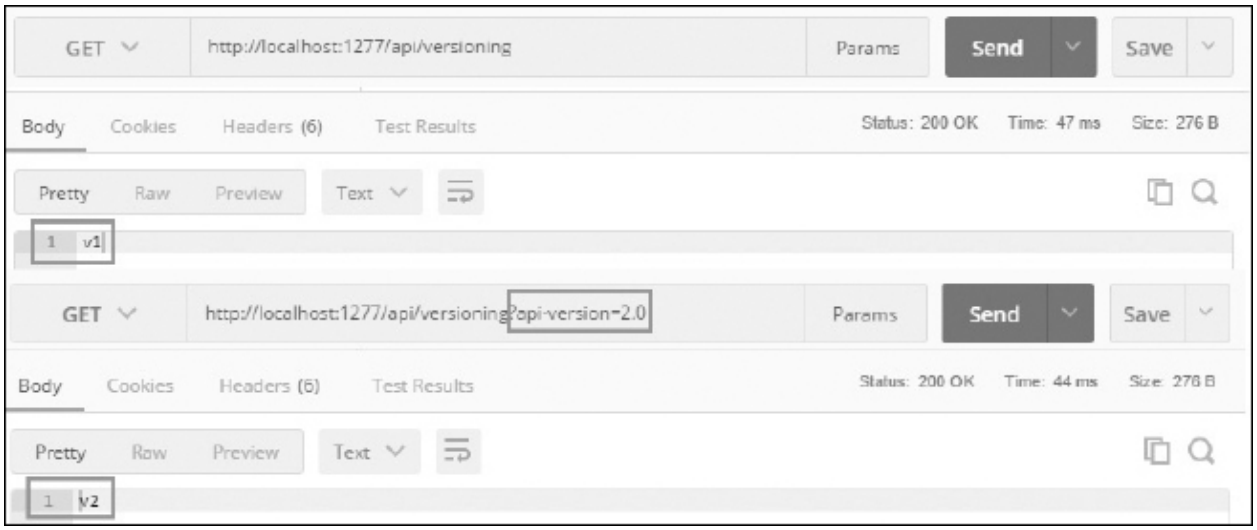


Figura 12.5 – La risposta alle API esposte sulla stessa route dipende dal parametro `api-version` specificato nella querystring.

In alternativa, come abbiamo anticipato, è possibile specificare la versione direttamente all’interno della route, anche se in questo caso particolare viene poi persa la possibilità di avere il fallback sulla versione di default. Poiché viene gestito a un livello differente, in caso in cui non venga più specificata la versione, il risultato dell’API ritornerà un errore 404 (Not Found). L’Esempio di codice 12.6 dimostra come sia fattibile semplicemente aggiornando il percorso della route con l’apposito attributo.

Esempio 12.6

```
[Route(”api/{version:apiVersion}/versioning”)]
```

Quando il codice inizia a evolvere in più versioni, talvolta può aver senso cominciare a deprecare endpoint che non sono più validi, anche se questi vengono ancora supportati e manutenuti. Per questo esiste la possibilità di aggiungere la proprietà `Deprecated`, che manderà in risposta un header `api-deprecated-version` contenente il numero della versione deprecata, come viene mostrato nell’Esempio 12.7.

Esempio 12.7

```
[ApiVersion(”1.0”, Deprecated = true)]
```

Al contrario, può succedere che una determinata API sia mantenuta invariata nel tempo anche con il passare delle versioni su altri endpoint: per questo, al posto di specificare ogni volta il supporto a una nuova versione, può far comodo specificare l’attributo `ApiVersionNeutral`.

Arrivati a questo punto, le API possono già essere utilizzate, distribuite e mantenute nel corso del tempo. Per via di come sono costruite le API, ovvero in una sorta di micro-servizi, può succedere talvolta che ci sia la necessità di doverle estenderle per realizzare un vero e proprio sistema complesso e personalizzato secondo le nostre esigenze: per questo introduciamo il concetto di WebHook.

Gestire endpoint basati su WebHook

Abbiamo già parlato durante questo stesso capitolo e nel capitolo precedente di cosa sono e di quanto le WebAPI possano essere utili per consumare dei servizi da parte di un client. Talvolta, però, può essere necessario estendere la funzionalità stessa esposta dal servizio sfruttando altri servizi di terze parti, ed è qui che vengono in gioco i WebHook: si tratta di chiamate effettuate con metodo `POST`, che vengono richiamate al verificarsi di un determinato evento, detto trigger.

I WebHook funzionano come una callback HTTP e sono supportati nativamente da ASP.NET Core a partire dalla versione 2.1, con il supporto per diversi provider come Azure Alerts, Kudu, Dynamics CRM, Bitbucket, Dropbox, GitHub, MailChimp, Pusher, Salesforce, Slack, Stripe, Trello e WordPress, con integrazioni per altri servizi previste in un futuro prossimo.

Negli esempi che seguiranno, vedremo l'integrazione tramite GitHub, dato che è probabilmente il servizio più conosciuto di quelli esposti in precedenza, ma il funzionamento è piuttosto simile nel caso degli altri provider. Per iniziare, è necessario installare il pacchetto di NuGet `Microsoft.AspNetCore.WebHooks.Receivers.GitHub` e registrare l'apposito servizio nella `ConfigureServices` della classe `Startup`, come è mostrato nell'Esempio 12.8.

Esempio 12.8

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc()
               .AddGitHubWebHooks();
}
```

Prima di ricevere le notifiche, l'altra cosa che bisogna fare all'interno del progetto ASP.NET Core è registrare una action che sia in grado di gestire la callback, come nel codice dell'Esempio 12.9.

Esempio 12.9

```
[GitHubWebHook]
public IActionResult Receiver(string id, string @event, JObject data)
{
   // ...
   return Ok();
}
```

Marcando la action `Receiver` con l'attributo `GitHubWebHook` facciamo in modo che sia in grado di gestire la route adeguata. I parametri esplicitati nella firma del metodo verranno riempiti con i valori che GitHub riterrà più opportuno e sono gli unici valori che possono variare tra WebHook esposti da provider differenti. GitHub potrà mandare una richiesta con un determinato payload riguardo a specifici eventi che avvengono su un repository specificato, come l'aggiunta di un nuovo commit, la creazione di un nuovo branch o di un nuovo tag, l'update di una issue o una nuova pull request, mentre per quanto riguarda Azure, per esempio, possiamo essere informati riguardo al superamento di una certa soglia di consumo: la funzione `Receiver` dovrà implementare la logica personalizzata secondo le esigenze del progetto e a seconda del trigger selezionato. All'interno delle impostazioni del repository di GitHub è possibile andare ad aggiungere un nuovo WebHook, come è illustrato nella Figura 12.6.

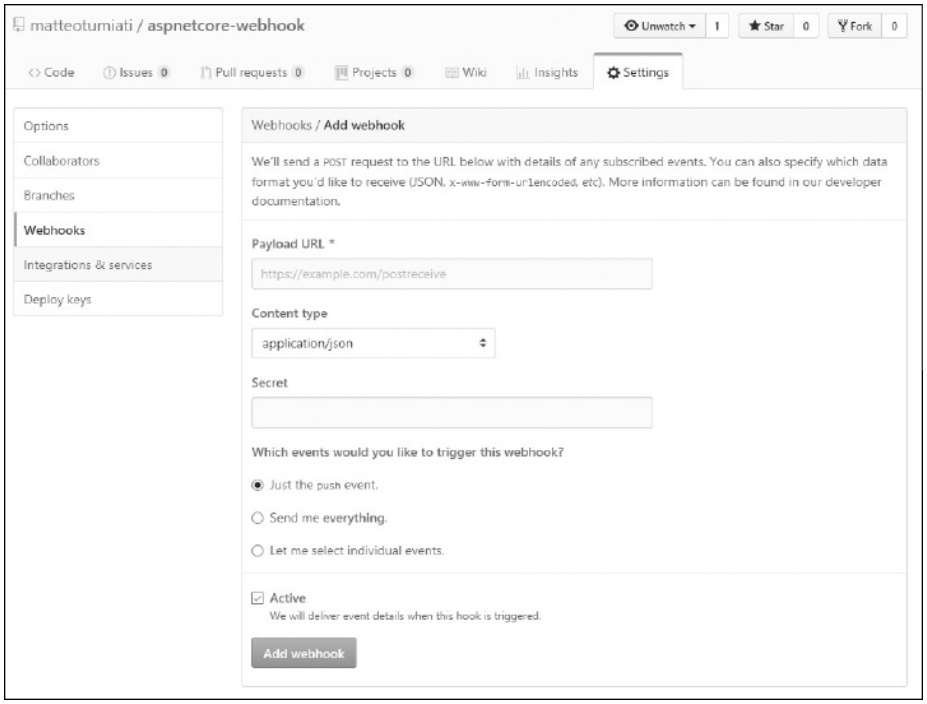


Figura 12.6 – Dal proprio repository di GitHub è possibile accedere alla configurazione dei WebHook tramite l'apposito menù all'interno delle impostazioni.

Poiché GitHub richiede un URL raggiungibile pubblicamente (e il localhost della macchina di sviluppo non lo è), dobbiamo affidarci a strumenti come ngrok, che permettono di esporre localhost su FQDN accessibili via internet. Per avviare ngrok è necessario lanciare dalla console di PowerShell il comando riportato nell'Esempio 12.10.

Esempio 12.10

```
.\ngrok.exe http localhost:{porta-esposta-da-aspnet-core}
```

Se l'ambiente richiesto è in grado di partire correttamente, il risultato ottenuto sarà simile a quello illustrato nella Figura 12.7.

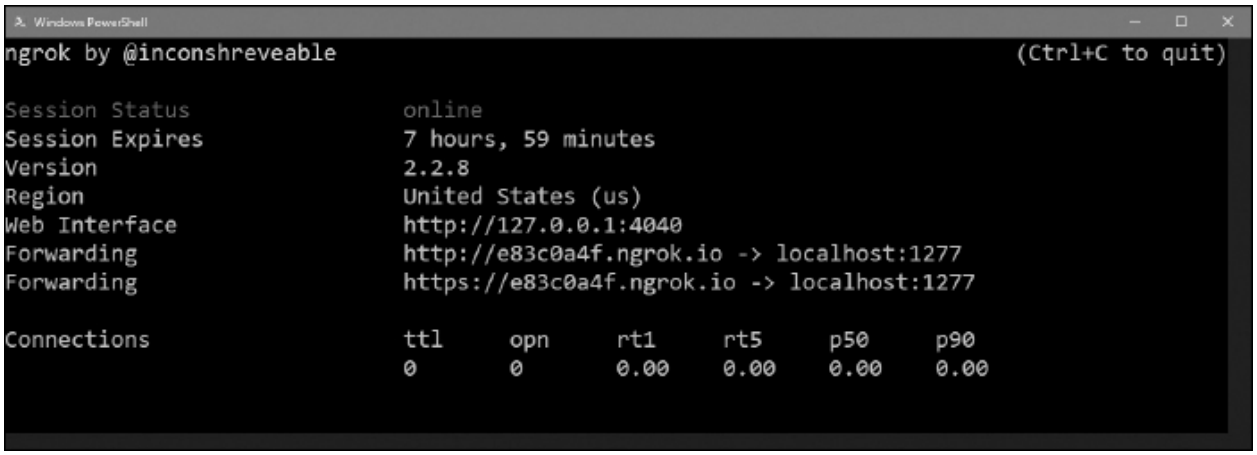


Figura 12.7 – Dopo aver lanciato il comando di avvio, tramite ngrok sarà possibile vedere gli URL generati con forwarding su HTTP e HTTPS, oltre che il periodo di uptime e il numero di connessioni attive.

A questo punto, possiamo modificare la configurazione di GitHub andando a impostare i parametri nel modo seguente:

- Payload URL: è l'host name assegnato da ngrok unito al path `/api/webhooks/incoming/github`.
- Content-Type: per via dei formatter impostati nel capitolo precedente è necessario impostare `application/json`.
- Secret: un token generato con un qualsiasi tool per HMAC.

Lo stesso secret deve essere poi importato anche all'interno del progetto nel file di `appsettings.json`, come è visibile nell'Esempio 12.11.

Esempio 12.11

```
"WebHooks": {
   "GitHub": {
      "SecretKey": {
         "default": "752a7cf3cd87c32b65a585b82fcf06ff9972f125"
      }
   }
}
```

Questa configurazione verrà poi letta in fase di startup dell'applicazione ASP.NET Core e sarà effettuata la registrazione del WebHook dedicato. Creando una issue all'interno del repository oppure facendo una nuova commit sul file, si potrà vedere, debuggando il progetto, il flusso di dati entrare all'interno della funzione definita in precedenza.

Al contrario di quanto viene introdotto dal concetto dei WebHook, ovvero che è bene fare separazione tra più servizi ed estenderli in un secondo momento, è bene notare come ci siano scenari in cui avere il dato nel minor tempo possibile è fondamentale per produrre immediatamente una notifica. Per questo, client e server devono essere fortemente accoppiati in modo da ridurre al massimo la latenza e c'è il bisogno di avere un framework, come SignalR, che semplifichi lo sviluppo.

Comunicazione in tempo reale con SignalR

All'interno di questo capitolo sono stati affrontati diversi meccanismi di comunicazione, partendo dalle WebAPI, asincrone e invocate su richiesta, ai WebHook, utilizzati principalmente per estendere il funzionamento di un determinato servizio attraverso una o più WebAPI. Nel capitolo precedente, invece, sono anche stati anche discussi quelli che sono i diversi vantaggi di avere una forte separazione tra il client e il server che espone il servizio, parlando per esempio di soluzioni basate su self-discovery ma, così come ci sono scenari in cui questo ha senso, ci sono altrettanti scenari in cui è necessario avere l'esatto opposto: ricerche o aggiornamenti di dati, le azioni e le operazioni finanziarie in genere, notifiche di un gol durante una partita, giochi online o applicazioni che richiedono della collaborazione in

tempo reale (per esempio Word Online) sono solo alcuni degli scenari che richiedono una latenza minima nello scambio dei messaggi. Sebbene si possa pensare che queste siano applicazioni molto specifiche, in realtà non lo sono; infatti, il concetto di notifica può essere applicato a qualsiasi ambiente, anche per applicazioni di campo industriale, per notificare l'avvenuta produzione di un determinato pezzo oppure l'aggiornamento sulla quantità dei prodotti venduti e così via. Inoltre, poiché principalmente il tutto viene riportato ad uso e consumo degli utenti, sorgono altri problemi: gli utenti vogliono vedere sempre tutto aggiornato, nel più breve tempo possibile e su ogni device con ogni tipo di connessione.

Il paradigma cambia quindi in modo drastico e di fatto non si parlerà più di client e server ma di publisher e subscriber: il subscriber, o consumer, è chi vuole iscriversi a un servizio esposto dal publisher per essere notificato all'avvenimento di un evento, mentre il publisher è colui che tiene traccia di tutti i consumer, che può identificare in modo univoco, in modo da interrogarli e di poter mandare loro i dati allo scatenarsi di un trigger, quale, per esempio, la modifica di un dato in un database, come si può vedere nell'architettura illustrata nella Figura 12.8.

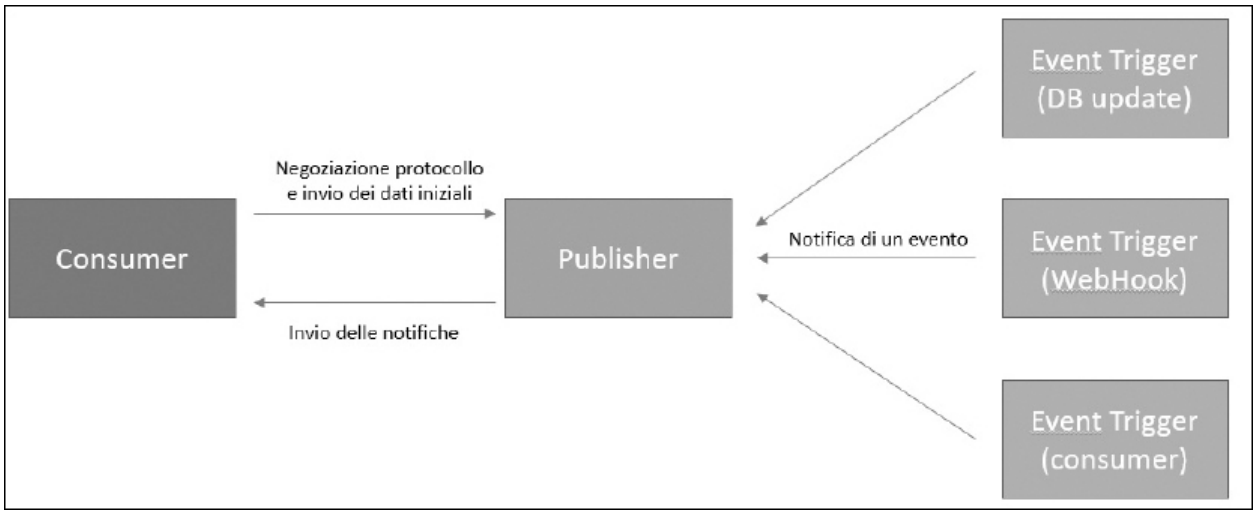


Figura 12.8 – L'architettura basata sul "pattern" di publisher e subscriber elimina di fatto la differenza tra client e server.

Le notifiche che vengono inviate dal publisher sono volatili, per cui non è previsto alcun sistema di persistenza o di invii multipli della stessa notifica qualora non ci sia un ack da parte del subscriber per notificare al subscriber la ricezione della notifica: la priorità è che la notifica deve arrivare nel più breve tempo possibile, altrimenti non avrebbe senso mandarla, per cui non c'è spazio per fare altre operazioni oltre all'invio del singolo messaggio. Nel caso in cui ci sia, invece, la necessità di persistere i messaggi, assicurarsi che arrivino a tutti i subscriber e che vengano gestiti tramite appositi ambienti "poisoned" quando non vengono prelevati dal consumer dopo un certo periodo temporale, allora è necessario puntare su meccanismi di code come Service Bus piuttosto che Azure Queue.

A livello di comunicazione, invece, sorge un nuovo problema poiché tutto, come abbiamo già visto, è attualmente basato su HTTP, un protocollo di request-response, non publisher-subscribe: come si può rendere compatibili questi due sistemi? Per dare una risposta a questa domanda sono nati diversi meccanismi e protocolli che funzionano al di sopra di HTTP:

- `Periodic polling`: il client chiede a intervalli regolari al server se ci sono novità riguardo a un determinato evento, e il server continua a rispondere con un payload indicante il risultato della richiesta effettuata dal client.

- `Long polling`: con l'introduzione di Comet, un modello architetturale basato sugli eventi server-side, non c'è più bisogno che il client richieda in modo costante aggiornamenti e che il server risponda a ogni richiesta, ma è direttamente il server che notifica al client il verificarsi di una determinata condizione; il client è comunque responsabile di effettuare nuovamente la richiesta in caso in cui ci siano timeout, ovvero quando il server dopo un certo periodo di tempo non ha ancora inviato una risposta in base ai dati richiesti dal client.

- `Server-Sent Events (SSE)`: sono uno standard definito in HTML5, basato su HTTP, che permette una comunicazione monodirezionale dal publisher verso i subscriber. Gli aggiornamenti vengono inviati in modo automatico, senza bisogno che il client effettui più volte la stessa richiesta.

- `WebSocket`: diventato standard W3C, consente la creazione di canali di comunicazione persistenti e bidirezionali attraverso una singola connessione TCP, garantendo un minor consumo di risorse da parte del subscriber. L'unica correlazione che ha con HTTP è durante la parte di handshake iniziale, in cui le due parti, il client e il server, stabiliscono se possono parlare entrambe con questo protocollo. Poi viene, a tutti gli effetti, aperta una socket e fatto l'upgrade della comunicazione.

Stabilire la tipologia di comunicazione ed eventualmente fare fallback da WebSocket, la migliore, da periodic-polling, la peggiore, non è un compito facile se dovessimo partire da zero. Date anche le premesse precedenti, potrebbe sembrare che realizzare un sistema di comunicazione in real-time sia quasi impossibile, ma in realtà ASP.NET Core include, in una parte dedicata del framework a partire da ASP.NET Core 2.1, un toolkit chiamato SignalR, che va ad astrarre gran parte della complessità, lasciando allo sviluppatore solo una piccola parte di configurazione e di logica, che è dipendente dal processo applicativo. Esattamente come per ASP.NET Core, anche SignalR è stato completamente riscritto da zero ma, nonostante questo, il team di sviluppo ha cercato di mantenere lo stesso approccio delle versioni per ASP.NET, in modo tale che gli sviluppatori non si trovino troppo in difficoltà a migrare una soluzione esistente.

SignalR è una libreria che si divide in due parti, che hanno entrambe lo scopo di fare processing asincrono dei dati in modo rapido: la parte server è in grado di mantenere le connessioni con tutti i consumer, di scalare il protocollo di comunicazione istantaneamente e di codificare i messaggi che deve scambiare, mentre la parte client è un set di librerie che permette la comunicazione con il server e che funziona su qualsiasi piattaforma, compreso il mobile nativo di Android e iOS. In particolare, rispetto alle versioni presenti su ASP.NET, c'è stato un grande passo in avanti per la parte client distribuita per le applicazioni web: costruita in TypeScript e distribuita tramite npm, garantisce maggiori performance, possibilità da parte di Microsoft di mantenerla e aggiornarla con più semplicità e, non meno importante, non ha più la dipendenza obbligatoria da jQuery, rendendola agnostica rispetto alle librerie JavaScript.

Iniziamo a vedere quanto sia facile lavorare, creando una prima connessione. Poiché, a partire da ASP.NET Core 2.1, SignalR è incluso nel framework, non è necessario aggiungere pacchetti di NuGet, ma è comunque necessario comunicare al server che si vuole fare uso del servizio, configurando, come al solito, il relativo middleware nello startup, come facciamo nell'Esempio 12.12.

Esempio 12.12

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddSignalR();
}
```

All'interno del metodo `Configure` invece, dobbiamo andare a registrare tutti gli `Hub`, ovvero tutte quelle classi che faranno da aggregatori dei messaggi che devono passare tra client e server e viceversa. Un hub, infatti, a livello concettuale non è nient'altro che lo stesso contenitore di azioni, un po' come il controller lo è per le action. La configurazione è contenuta nell'Esempio 12.13.

Esempio 12.13

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   app.UseSignalR(routes =>
   {
      routes.MapHub<ChatHub>("/chat");
      routes.MapHub<StreamingHub>("/streaming");
   });
}
```

In questo esempio mappiamo su due route ben specifiche altrettanti hub. Le route sono completamente personalizzabili e un ciascun hub è una classe che eredita dalla classe base `Hub` di SignalR, che si comporta da aggregatore e consente di rimandare i messaggi con le varie politiche del caso: a tutti, in broadcast, a un client ben specifico, individuato da un `ConnectionId`, oppure a un gruppo specifico di utenti. Nell'Esempio 12.14 viene creato un `ChatHub`, ovvero un `Hub` che permette di gestire una chat tra utenti.

Esempio 12.14

```
public class ChatHub : Hub
{
   public void Send(string name, string message)
   {
      Clients.All.SendAsync("broadcastMessage", name, message);
   }
   public override Task OnConnectedAsync()
   {
      Clients.All.SendAsync("broadcastMessage", "system", $"{Context. ConnectionId} è entrato nella
                           chat.");
      return base.OnConnectedAsync();
   }
   public override Task OnDisconnectedAsync(Exception exception)
   {
      Clients.All.SendAsync("broadcastMessage", "system", $"{Context. ConnectionId} è uscito dalla
                           chat.");
      return base.OnDisconnectedAsync(exception);
   }
}
```

I metodi OnConnectedAsync e OnDisconnectedAsync vengono utilizzati per comunicare a tutti gli utenti in ascolto, in modalità broadcast, che un determinato utente, identificato da un suo ConnectionId, si è collegato (oppure disconnesso) alla chat corrente. La funzione di Send invece, non è un override di un metodo base esposto dalla classe Hub di SignalR, ma, all'opposto, è una funzione costruita per i nostri scopi: in questo caso il server, così come il client, potrà invocare la funzione Send definita nel ChatHub per notificare a tutti i client che l'utente, definito dalla proprietà name, ha inviato il messaggio message. Saranno poi i vari client a doversi registrare, con una tecnica di tipo publisher-subscribe, all'evento di broadcastMessage, così da essere notificati dell'invio di un messaggio nella chat da parte di qualche altro utente.

Il subscriber che andrà a collegarsi può essere una qualsiasi applicazione desktop, per esempio WPF, piuttosto che un'app mobile, oltre che un'applicazione web. La parte grafica della chat che verrà realizzata è composta da questa scarsa porzione di HTML, mostrata nell'Esempio 12.15.

Esempio 12.15

```
<body>
   <div class="container">
      <input type="text" id="message" class="message" />
      <input type="button" id="sendmessage" value="Send" class="btn" />
      <ul id="discussion"></ul>
   </div>
   <script type="text/javascript" src="scripts/signalr-client.js"></script>
   <script type="text/javascript" src="scripts/chat.js"></script>
</body>
```

Il codice è costituito da una semplice form – in cui è contenuta una casella di testo, che rappresenta il messaggio da inviare a tutti i client – e dal pulsante sendmessage, che chiamerà la funzione Send definita lato server per notificare tutti gli altri client. Viene anche aggiunta una lista vuota, chiamata discussion, in cui verranno elencati tutti i messaggi ricevuti sotto forma di elenco puntato. La definizione di SignalR lato client è contenuta nel file signalr-client.js, scaricato tramite il comando npm install @aspnet/signalr.

Allo stesso tempo, va anche definita tutta la logica applicativa, contenuta nel file chat.js, esposta nell'Esempio 12.16.

Esempio 12.16

```
document.addEventListener('DOMContentLoaded', function () {

   // recupero il messaggio che voglio inviare
   var messageInput = document.getElementById('message');
   // chiedo all'utente il suo nome (utilizzato nella chat).
   var name = prompt('Inserisci il tuo nome:', '');
   messageInput.focus();
   // apro la connessione sul ChatHub
   startConnection('/chat', function (connection) {
      // creo la funzione 'broadcastMessage' invocata dal 'ChatHub'
      connection.on('broadcastMessage', function (name, message) {
         // aggiungo il messaggio come parte dell'elemento 'discussion'
         var liElement = document.createElement('li');
         liElement.innerHTML = '<strong>' + name + '</strong>:  ' + message;
         document.getElementById('discussion').appendChild(liElement);
      });
   }).then(function (connection) {
      console.log('connection started');
      // recupero il click del pulsante per inviare il messaggio
      document.getElementById('sendmessage').addEventListener('click', function (event) {

         // chiamo il metodo 'Send' all'interno del 'ChatHub'
         connection.invoke('send', name, messageInput.value);
         // resetto i valori di input
         messageInput.value = '';
         messageInput.focus();
         event.preventDefault();
      });
   }).catch(error => {
      console.error(error.message);
   });
   function startConnection(url, configureConnection) {
      return function start(transport) {
```

```
         console.log('Starting connection using ${signalR.TransportType[transport]} transport');
         var connection = new signalR.HubConnection(url, { transport: transport });

         if (configureConnection && typeof configureConnection === 'function') {
            configureConnection(connection);
         }
         return connection.start()
            .then(function () {
               return connection;
         })
            .catch(function (error) {
               console.log('Cannot start the connection use ${signalR.TransportType[transport]}
               transport. ${error.message}');

               if (transport !== signalR.TransportType.LongPolling) {
                  return start(transport + 1);
               }

               return Promise.reject(error);
            });
      }(signalR.TransportType.WebSockets);
   }
});
```

La logica del codice JavaScript consiste nell'aspettare che venga caricato il DOM della pagina, quindi viene chiesto all'utente, tramite una dialog, il suo nome: questo valore verrà salvato per essere poi rimandato nella proprietà `name` al server. Viene quindi recuperato il messaggio e viene avviata la connessione sullo hub `ChatHub`, che può avvenire in diverse modalità: la prima, la più diffusa tra i browser moderni, è basata sul protocollo WebSocket ma, come abbiano anticipato, in caso di non funzionamento perché non supportata, la logica è in grado di fare fallback su Server-Sent Events (SSE) e long polling, prima di rifiutare la connessione, poiché SignalR è in grado di fare protocol negotiation con il client.

Una volta ottenuto un collegamento verso il server con una delle modalità appena descritte, diventa fondamentale registrarsi all'evento di click del pulsante della form: a ogni pressione, infatti, verrà recuperato il messaggio scritto dall'utente, che verrà rimandato, assieme al nome dell'utente, alla funzione `Send` definita lato server tramite una chiamata a `invoke` dell'oggetto `connection` ottenuto in precedenza. Poiché lato server il messaggio viene inviato a tutti i client in modo indifferente senza escludere nessuno, anche il sender stesso del messaggio lo riceverà. Per questo, grazie alla funzione `on` richiamata sull'oggetto `connection`, ci possiamo mettere in ascolto di un messaggio inviato dalla funzione `broadcastMessage` chiamata dal server. Alla ricezione dei dati, verrà semplicemente aggiunto un nuovo punto all'elenco puntato definito sulla form come `discussion`.

Una delle novità di SignalR per ASP.NET Core rispetto al passato è il supporto per lo streaming: grazie alle performance raggiunte con la nuova infrastruttura, è possibile inviare dati da server a client prima dell'invocazione sull'hub, con un numero decisamente maggiore di client rispetto al passato, che può essere collegato contemporaneamente. Vediamo ora un esempio di streaming, definendo uno hub come quello dell'Esempio 12.17.

Esempio 12.17

```
public class StreamingHub : Hub
{
   public IObservable<string> StartStreaming()
   {
      return Observable.Create(
         async (IObserver<string> observer) =>
         {
            for (var i = 0; ; i++)
            {
               observer.OnNext($"Invio messaggio {i}...");
               await Task.Delay(1000);
            }
         });
   }
}
```

Al contrario del `ChatHub` mostrato in precedenza, seguendo questa nuova tecnica non è più importante registrarsi agli eventi di connessione e disconnessione del client, perché abbiamo solo un metodo chiamato `StartStreaming` che apre, attraverso l'uso di `System.Reactive.Linq`, un oggetto di tipo `Observable`. Il funzionamento consiste solamente nell'invio di un messaggio su tutti i subscriber registrati all'oggetto `Observable`, all'infinito e con un ritardo temporale tra l'invio di un messaggio e il successivo di circa un secondo.

Il client, che per fare una variazione sul tema sarà una console application .NET Core, avrà solamente la necessità di aggiungere una reference al relativo pacchetto di NuGet `Microsoft.AspNetCore.SignalR.Client` e implementare poche righe di codice per leggere i dati in tempo reale. L'Esempio 12.18 contiene il codice necessario a implementare questa funzionalità.

Esempio 12.18

```
static async Task MainAsync(string[] args)
{
   var streamingConnection = new HubConnectionBuilder()
                     .WithUrl("http://localhost:5000/streaming")
                     .WithConsoleLogger()
                     .Build();

   // apertura della connessione
   await streamingConnection.StartAsync();
   // apertura del canale per lo streaming sulla funzione StartStreaming
   var channel = await _streamingConnection.StreamAsChannelAsync<string>("StartS treaming");

   // si aspetta l'arrivo di un messaggio
   while (await channel.WaitToReadAsync())
   {
      // lettura del messaggio e stampa su console
      while (channel.TryRead(out string message)
         Console.WriteLine($"Messaggio ricevuto: {message}");
   }
}
```

La prima cosa fatta è stata creare un oggetto di tipo `HubConnectionBuilder` per aprire la connessione verso la route dov'è definito lo `StreamingHub` lato server, per poi avviare la connessione con la chiamata al metodo asincrono `StartAsync` e, infine, abbiamo creato l'oggetto `channel` sul quale ascoltiamo messaggi provenienti dalla funzione `StartStreaming` definita lato server. Ogni messaggio inviato dal server verrà letto dal client in modalità asincrona e quindi scritto sulla console. Tutta la complessità definita con le precedenti versioni di SignalR si è ridotta all'osso e, comparata con altri framework analoghi, questa opzione garantisce sicuramente ottime prestazioni nonostante sia un prodotto relativamente nuovo e costruito da zero.

Quello che è dato per scontato e di cui non abbiamo ancora discusso riguarda come i dati vengono serializzati per poter essere inviati come messaggio dal publisher al subscriber e viceversa: di default c'è il supporto al classico text-based con messaggi in JSON, ma SignalR per ASP.NET Core introduce anche il supporto per MessagePack, che effettua la serializzazione in formato binario, in modo da garantire messaggi di dimensioni inferiori e più veloci da spedire. MessagePack non è integrato nel framework ma è distribuito come pacchetto di NuGet `Microsoft.AspNetCore.SignalR.MsgPack`. L'aggiunta di questo formato è illustrata nell'Esempio 12.19.

Esempio 12.19

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddSignalR()
            .AddMessagePackProtocol();
}
```

Per la parte client, invece, è necessario aggiungere il pacchetto di NuGet `Microsoft.AspNetCore.SignalR.Client.MsgPack` e specificare il protocollo di serializzazione in fase di collegamento all'hub, come è illustrato nell'Esempio 12.20.

Esempio 12.20

```
_var streamingConnection = new HubConnectionBuilder()
                     .WithUrl("http://localhost:5000/streaming")
                     .WithMessagePackProtocol()
                     .WithConsoleLogger()
.Build();
```

Per maggiori informazioni sul funzionamento del protocollo MessagePack e per capire se è supportato dalla nostra architettura, rimandiamo alla documentazione ufficiale disponibile all'indirizzo: http://aspit.co/bnn.

Conclusioni

All'interno di questo capitolo sono stati affrontati dei temi avanzati rispetto alla creazione di una WebAPI classica e, in particolare, si è parlato di tutto ciò che può essere utile per riuscire a mantenere il servizio nel tempo, trattando in particolare gli scenari relativi alla documentazione con Swagger e al versionamento. Proseguendo all'interno del capitolo, si è visto come fare uso dei WebHook, ovvero di una sorta di callback HTTP, per estendere le capacità delle API distribuite su più livelli, in modo che possano lavorare ed evolvere indipendenti gli uni dagli altri, garantendo in un qualche modo la continuità di servizio. Al contrario, invece, abbiamo visto che esistono sistemi che hanno forti necessità di comunicare in tempo reale, e per questi non c'è modo di avere una separazione tra il client e il server,

poiché entrambi devono potersi “conoscere” e “parlare” nel più breve tempo possibile e con messaggi che siano i più piccoli possibili, pertanto si è discusso della nuova versione di SignalR, integrata in ASP.NET Core, e di come vada a implementare, rispetto al passato, un paradigma decisamente più moderno e vicino agli sviluppatori, che nella sua ultima versione raggiunge performance notevoli, garantendo la possibilità di lavorare in streaming, ma consentendo anche l’indipendenza da pacchetti esterni come jQuery.

Negli esempi affrontati nel corso del capitolo, l’ambiente era controllato, pertanto la generazione di qualche eccezione era principalmente voluta ma, nell’ambiente di produzione, quando si rilasciano questa ed altre funzionalità, non è proprio quello che ci si aspetta. Per questo è necessario iniziare a parlare, come vedremo nel prossimo capitolo, di strumenti legati alla diagnostica e alle performance, per capire nel più breve tempo possibile cosa eventualmente sta generando (o genererà) problemi nelle nostre applicazioni, per essere in grado di intervenire e risolverli.

13
Gestione e diagnostica degli errori

I capitoli precedenti sono sufficienti per capire e sviluppare un applicativo completo che sappia gestire form o fornire servizi REST, ma una soluzione ben fatta deve saper gestire anche le situazioni impreviste, dare messaggi di errore amichevoli agli utenti e raccogliere informazioni utili a una eventuale diagnostica. In caso di errore dobbiamo accorgercene o, in caso di malfunzionamento segnalato dall'utente, dobbiamo poter ricostruire la situazione che si è verificata, capire lo stato dei dati o identificare logiche errate che non necessariamente causano un errore bloccante all'utente.

In questo capitolo vogliamo mostrare tutti gli strumenti inclusi in ASP.NET Core per poter gestire gli errori, tracciare tutto ciò che avviene ed eventualmente appoggiarci a servizi terzi, come quelli forniti da Microsoft Azure, per avere statistiche o capire l'impatto di un errore.

La gestione degli errori a runtime

Quando parliamo di errore, stiamo in realtà generalizzando, perché in un applicativo gli errori possono essere di natura diversa, con livelli diversi. A fronte di una richiesta fatta dall'utente, la risposta che può giungere all'utente può essere un messaggio di errore generico, uguale per tutte le pagine, oppure mirato a fornire informazioni sull'operazione appena fatta dall'utente, ma che non ha raggiunto gli scopi prefissati. Ne è un esempio la validazione della form vista nel Capitolo 7: la pagina appena inviata viene mostrata nuovamente all'utente, ma fornendo messaggi di errore su ogni singolo campo o sull'intera form. Questo genere di comportamento è da preferire in ogni situazione, perciò in ogni pagina e azione dobbiamo sempre domandarci cosa potrebbe andare storto: il codice contempla variabili nulle o vuote? Vengono gestite possibili eccezioni? Quest'ultima domanda è sempre la più difficile alla quale rispondere, ma non c'è dubbio che, in scenari cloud e sempre più distribuiti, una chiamata a un database o a un servizio web può, in alcuni casi, andare in errore, e questa situazione va prevista.

Nei nostri controller e nelle nostre pagine è opportuno quindi utilizzare il costrutto `try/catch` per intercettare questo genere di eccezioni e mostrare un messaggio di errore amichevole all'utente. È inutile, infatti, dare dettagli tecnici, ed è da preferire l'uso di messaggi più generici.

Il `ModelState`, utilizzato per popolare gli errori del modello, è un buon contenitore per inserire anche errori generici, come viene mostrato nell'Esempio 13.1.

Esempio 13.1

```
public void OnPost()
{
   try
   {
      // Codice che chiama dei servizi...
   }
   catch (WebException)
   {
      // Aggiungo un errore generico al modello
      ModelState.AddModelError("",
         "Errore nella chiamata ai servizi, riprovare.");
   }
}
```

Il primo parametro della funzione `AddModelError` indica il nome della proprietà del modello alla quale associare l'errore. Se non specificato, l'errore è generico e può essere facilmente visualizzato nella view attraverso il tag helper, come è mostrato nell'Esempio 13.2.

Esempio 13.2

```
<form asp-page="Index">
<div asp-validation-summary="ModelOnly"></div>
```

L'utilizzo dell'enumerato `ModelOnly` forza la visualizzazione di errori generici del modello ed è facoltativo.

Gli errori durante lo sviluppo

Per quanto riguarda, invece, gli altri tipi di errore, cioè quelli che non prevediamo o che sono frutto di codice non scritto correttamente, possiamo ricorrere a una gestione generica che copra l'intero applicativo. ASP.NET Core è in grado di intercettare eccezioni generate all'interno della richiesta e, come prevede il protocollo HTTP, risponde con uno status code 500 visualizzato con una pagina generica di errore del browser, di poca utilità per l'utente finale, anche in caso si richieste REST.

Abbiamo quindi già visto nel Capitolo 3 che il template di Visual Studio predispone la classe `Startup` con la configurazione di due middleware, come viene mostrato nell'Esempio 13.3.

Esempio 13.3

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
   }
   else
   {
      app.UseExceptionHandler("/Error");
}
```

I due middleware differenziano il comportamento da adottare nel caso in cui ci troviamo nell'ambiente di sviluppo o di produzione. `UseDeveloperExceptionPage` installa un middleware che intercetta le eccezioni generate e produce un HTML utile a noi sviluppatori per diagnosticare l'errore. Questo middleware non va abilitato in scenari di produzione perché, oltre a non fornire un'interfaccia consona all'utente, può dare informazioni sensibili o utili a un eventuale hacker che vuole capirne il funzionamento.

L'extension method contiene un overload al quale è possibile passare un oggetto di tipo `DeveloperExceptionPageOptions` che dispone di due proprietà: `SourceCodeLineCount` e `FileProvider`. Il primo ci permette di indicare quante righe della sorgente possiamo visualizzare, mentre il secondo ci permette di personalizzare i provider da utilizzare per leggere il file sorgente, necessario solo se il codice non è nostro.

Un secondo middleware, utile sempre per la fase di sviluppo, è configurabile tramite l'extension method `UseDatabaseErrorPage`, come viene mostrato nell'Esempio 13.4.

Esempio 13.4

```
if (env.IsDevelopment())
{
   app.UseDeveloperExceptionPage();
   app.UseDatabaseErrorPage();
}
```

Intercetta le eccezioni relative a Entity Framework Core e, in particolare, quelle riguardanti la migrazione della struttura del database. Qualora siano pendenti una o più migrazioni, una pagina web suggerisce l'operazione da fare, come viene mostrato nella Figura 13.1.

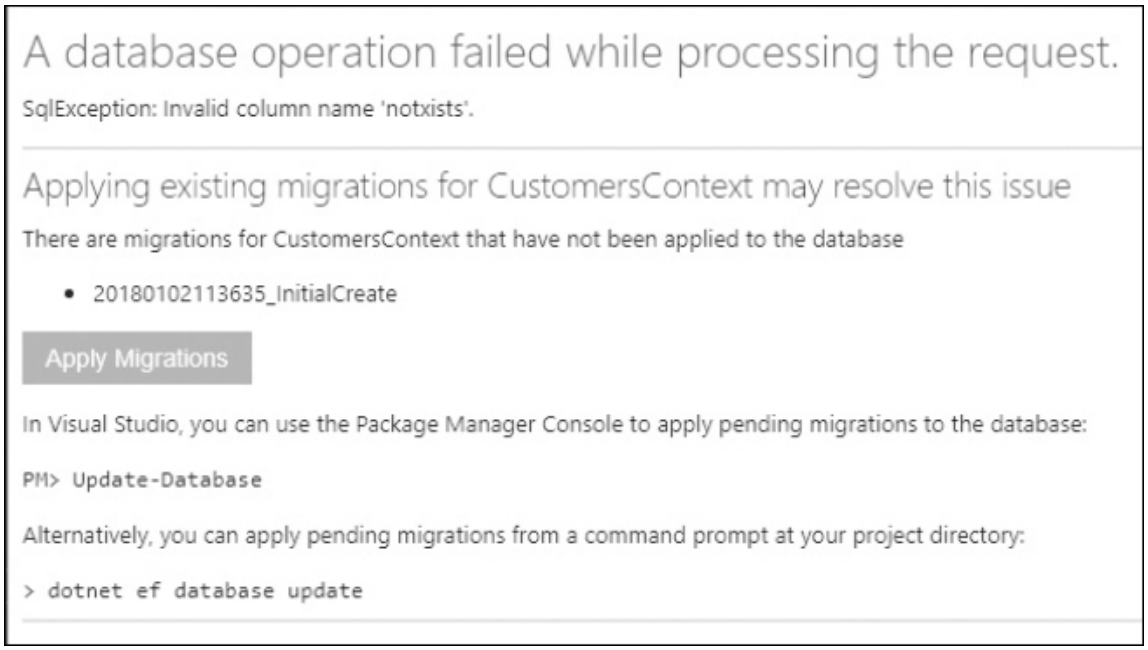


Figura 13.1 – Pagina di errore in caso di migrazione pendente di Entity Framework.

Premendo il pulsante non facciamo altro che attivare la procedura di update, la stessa che sarebbe possibile avviare via codice o utilizzando DotNet CLI. Questo middleware va di conseguenza adottato se usiamo il meccanismo delle migrazioni e solo per facilitare la loro applicazione. Inoltre, è di fondamentale importanza la configurazione del middleware fatta nell'Esempio 13.4, effettuata successivamente a quella di gestione generica degli errori.

Gli errori per l'utente

Riprendiamo ora l'Esempio 13.3 e la chiamata al metodo `UseExceptionHandler`, un ulteriore middleware da attivare per gli scenari di produzione. Esso intercetta tutte le eccezioni generate nel nostro codice server e fornisce una pagina di errore standard per gestirle. Nel template generato da Visual Studio, troviamo l'uso del percorso `/Error`, una pagina implementata attraverso una action di MVC o una Razor Page, a seconda della tipologia di progetto scelto.

> *Questi extension method per la gestione degli errori devono essere chiamati prima di tutti gli altri dedicati alla configurazione dei middleware. Fungono da contenitori per la restante parte della pipeline e ne intercettano eventuali errori.*

Il suo aspetto predefinito è visibile nella Figura 13.2.

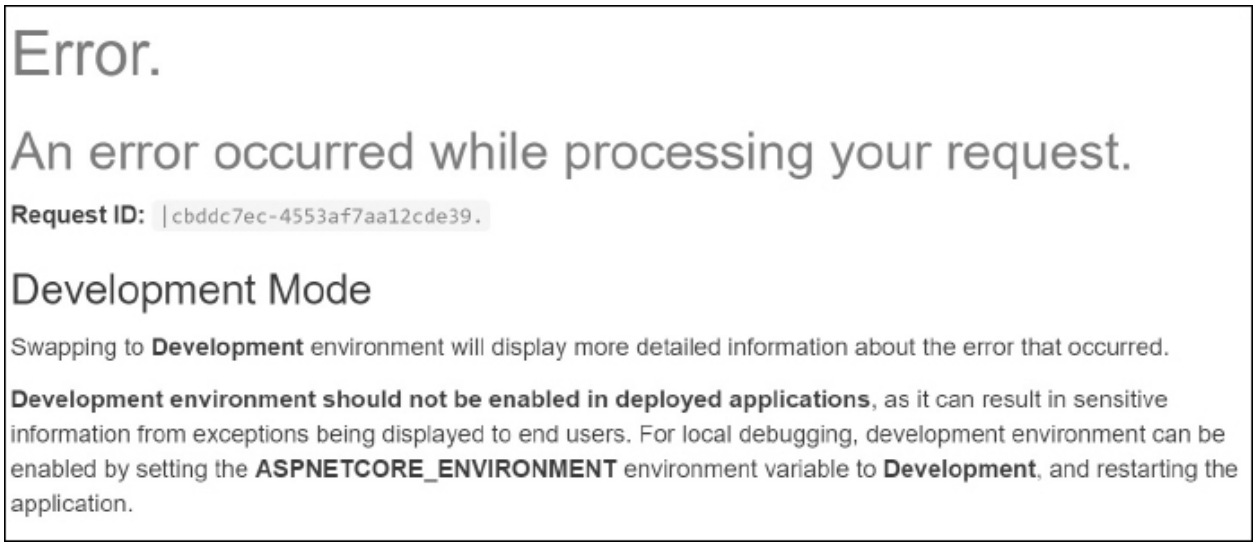


Figura 13.2 – Pagina di errore standard per l'utente.

Tra le cose da fare prima della messa in produzione, c'è sicuramente la personalizzazione di questa pagina per mostrare un messaggio di errore nella lingua corretta e senza alcuni aspetti tecnici che non riguardano l'utente. Inoltre, possiamo scegliere di cambiare il percorso della pagina a seconda delle nostre esigenze.

Oltre a intervenire sull'HTML della view, possiamo modificare il modello che sfruttiamo per dare informazioni all'utente, simile all'Esempio 13.5.

Esempio 13.5

```
public class ErrorModel : PageModel
{
   public string RequestId { get; set; }
   public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);
   public void OnGet()
   {
      RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
   }
}
```

La proprietà `RequestId` viene messa a disposizione per mostrare l'identificativo della richiesta corrente, ottenuta dal sistema di diagnostica di .NET o l'identificativo fornito dall'infrastruttura di ASP.NET Core. Questo codice, come vedremo più avanti nel capitolo, può essere mostrato all'utente per consentirgli eventualmente di effettuare segnalazioni e con esso darci la possibilità di correlare un eventuale logging all'errore che si è verificato, metodo decisamente più efficace rispetto a identificare il contesto di nostro interesse ricercandolo in un intervallo temporale.

La personalizzazione può avvenire anche sfruttando alcune informazioni aggiuntive che vengono aggiunte nel contesto e, in particolare, l'eccezione che ha portato alla visualizzazione della pagina di errore. Attraverso la collezione `Features` dell'oggetto `HttpContext` possiamo accedere a quella di tipo `IExceptionHandlerFeature` contenente la proprietà `Error`. Come riportato nell'Esempio 13.6, sfruttiamo questa informazione per cambiare il messaggio da mostrare all'utente.

Esempio 13.6

```
public string ErrorMessage
{
   get
   {
      // Accedo alla feature
      var exceptionHandlerFeature =
         HttpContext.Features.Get<IExceptionHandlerFeature>();
      // Personalizzo il messaggio da mostrare
      switch (exceptionHandlerFeature?.Error)
      {
         case WebException we:
            return "Errore nella chiamata ai servizi. Riprovare";
         default:
            return "Si è verificato un errore non previsto. Riprovare";
      }
   }
}
```

Questa proprietà può essere poi visualizzata nella view attraverso Razor, per fornire un'interfaccia più ricca.

Dobbiamo sottolineare che quando si verifica un errore e la pagina specifica viene visualizzata, l'indirizzo della pagina è ancora quello che ha generato l'eccezione. L'utente non subisce nessun redirect HTTP, ma vengono processate due pagine: la prima, che generato il problema, e la seconda, che mostra il messaggio.

Possiamo in alternativa usare l'overload dell'extension method `UseExceptionHandler` che accetta una funzione in grado di configurare un `IApplicationBuilder`, un po' come già facciamo sulla funzione `Configure` dello `Startup`. L'unica differenza è che in essa dobbiamo configurare i middleware da richiamare in caso di errore. Nell'Esempio 13.7 sfruttiamo questa possibilità tramite un middleware basato su lambda per rimandare l'utente a una pagina di errore, questa volta però tramite redirect HTTP.

Esempio 13.7

```
app.UseExceptionHandler(b => b.Use((context, next) =>
{
   context.Response.Redirect("/Error");
   // Nessuna operazione asincrona
   return Task.CompletedTask;
}));
```

Così facendo abbiamo il pieno controllo sulla risposta da dare all'utente anche se tramite il redirect perdiamo l'informazione sull'errore accessibile tramite feature, poiché appartenente alla richiesta che ha generato l'errore e non alla seconda dove l'utente è stato rimandato. Questo overload può essere utile anche nel caso in cui vogliamo fornire una risposta JSON a fronte di una richiesta REST, per la quale una risposta HTML non sarebbe adeguata. Gli errori delle nostre pagine e gli status code 500, però, non sono gli unici dei quali ci dobbiamo preoccupare.

Le pagine per gli status code

Tra gli status code più comuni in cui l'utente si può imbattere c'è sicuramente il 400, per una richiesta errata, un 404 per una pagina non trovata o un 403 per un accesso vietato. Questi codici, che vanno dal 400 al 599, possono essere generati da qualsiasi middleware: da una action di MVC, da una funzione di una Razor Page, da una web API o da un file statico. Quando si verificano, anche in questo caso, il browser mostra una pagina bianca.

A questo scopo vengono in aiuto altri middleware. Il più semplice si configura con l'extension method `UseStatusCodePages` il quale, a fronte di una risposta con i codici prima citati, produce il semplice HTML visibile nella Figura 13.3.

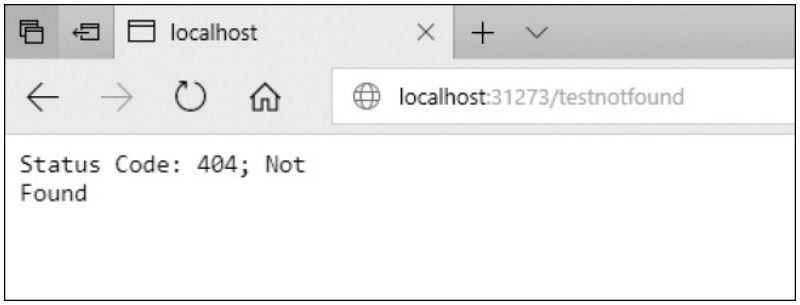


Figura 13.3 – Pagina predefinita per lo status code 404.

Possiamo configurare il middleware affinché utilizzi una nostra funzione o un nostro middleware, in maniera del tutto identica a quanto visto nell'Esempio 13.7, ma il metodo migliore per produrre pagine HTML di più elevata qualità è sicuramente quello di affidarsi ad altri due middleware: `UseStatusCodePagesWithRedirects` e `UseStatusCodePagesWithReExecute`. Il primo effettua un redirect HTTP, mentre il secondo una esecuzione all'interno della richiesta stessa, come fa `UseExceptionHandler`. Utilizziamo quindi quest'ultimo per realizzare una view tramite Razor Page, come mostrato nell'Esempio 13.8.

Esempio 13.8

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   // Gestione errori
   app.UseExceptionHandler("/error")
   // Personalizzazione degli status code
   app.UseStatusCodePagesWithReExecute("/status/{0}");
   // Altro...
app.UseStaticFiles();
```

L'utilizzo del placeholder "{0}" ci permette di inserire dinamicamente il codice all'interno del percorso e quindi implementare in un unico punto la pagina di stato.

Possiamo implementare questa semplice pagina, per esempio, con una Razor Page con una regola di routing personalizzata, in modo da ricevere lo status code nel percorso indicato, come viene mostrato nell'Esempio 13.9.

Esempio 13.9

```
@page "{code}"
@using Microsoft.AspNetCore.Routing
@{
   string code = HttpContext.GetRouteValue("code").ToString();
   ViewBag.Title = code;
}
<style>
   .big {
      font-size: 200pt;
      line-height: 0;
      color: #ccc
   }
</style>
<h1 class="big">@code</h1>
```

Nella view è importante notare il placeholder di nome `code` e la lettura del suo valore a runtime. Il codice è piuttosto semplice, perciò non abbiamo definito un modello apposito ma tutta la logica è stata definita nella view stessa, il cui risultato è visibile nella Figura 13.4.

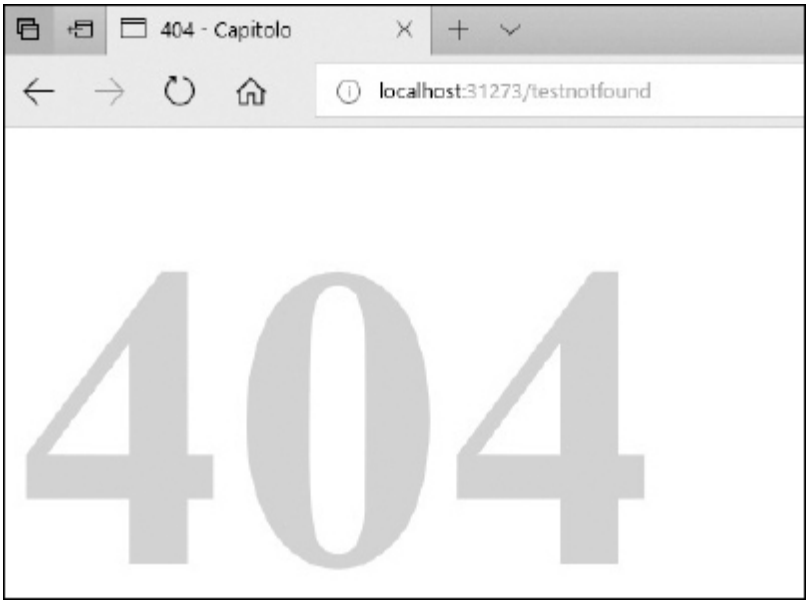


Figura 13.4 – Pagina personalizzata per lo status code 404.

Vi sono, infine, delle situazioni in cui non vogliamo che il middleware in questione personalizzi la risposta ma vogliamo forzare il comportamento predefinito. Anche in questo caso viene in aiuto una feature di nome `IStatusCodePagesFeature`, che tramite la sua proprietà `Enabled` ci permette di disabilitare per la richiesta corrente il middleware della status page, come mostrato nell'Esempio 13.10.

Esempio 13.10

```
public IActionResult TestNotFound()
{
      // Disattivo il middleware per gli status page
      var statusCodePagesFeature = HttpContext.Features
         .Get<IStatusCodePagesFeature>();
      statusCodePagesFeature.Enabled = false;
      return NotFound();
}
```

Nell'esempio viene disattivata l'opzione durante una action di un controller, prima che restituisca un 404. Dopo aver visto come gestire i messaggi di errore e le informazioni nei confronti dell'utente, vediamo ora come possiamo raccogliere informazioni utili alla diagnostica.

Tracciare gli eventi

Nel Capitolo 3 abbiamo introdotto i due servizi `ILogger` e `ILoggerFactory`, con i quali possiamo tracciare eventi all'interno del nostro applicativo. Tramite la dependency injection, possiamo sfruttare questi oggetti all'interno di tutti i layer applicativi, dal presentation costituito da ASP.NET Core, fino ad arrivare all'accesso ai dati o a un canale di comunicazione. Il namespace `Microsoft.Extensions.Logging` e l'omonimo pacchetto NuGet compatibile .NET Standard rendono questi componenti indipendenti e utilizzabili anche su altri runtime.

Attraverso un'istanza di `ILogger` possiamo tracciare qualsiasi stringa di testo a noi utile per capire cosa è successo, perciò è fondamentale essere più descrittivi e ricchi nelle informazioni fornite. Riprendiamo un esempio su come tracciare un'informazione durante una chiamata a una action di un controller MVC.

Esempio 13.11

```
public class LoggingController : Controller
{
   private readonly ILogger<LoggingController> _logger;
   public LoggingController(ILogger<LoggingController> logger)
   {
      _logger = logger;
   }
   public IActionResult Index(string mode, int type)
   {
      _logger.LogInformation("Index with {modeValue} and {typeValue}",
      mode, type);
   return View();
}
```

La funzione `LogInformation` accetta un messaggio e, facoltativamente, uno o più parametri. Diversamente da quanto potremmo pensare, il messaggio supporta una sintassi di template ma non usa la string interpolation di C#. I segnaposto messi tra graffe di nome `modeValue` e `typeValue` indicano i nomi dei segnaposto stessi, mentre il loro rispettivo valore viene assegnato in modo posizionale con i parametri `mode` e `type`, passati alla funzione. È di fondamentale importanza, quindi, l'ordine dei segnaposto e dei parametri passati. Diversamente dai tradizionali sistemi di log, il tracciamento viene effettuato in modo strutturato, mantenendo separati il messaggio e i parametri. Spetta poi al provider che lo mostra o lo persiste se decidere di formattare il template o di memorizzare separatamente i valori.

> *Poiché il tracciamento viene effettuato in maniera strutturata per poi essere memorizzato, è opportuno dosare i parametri, che possono essere anche oggetti complessi, per evitare di avere database spropositati con informazioni superflue.*

Questo modo di tracciare migliora la qualità dell'informazione e la sua ricerca, decisamente più accurata rispetto a una semplice analisi testuale.

Sebbene più informazioni tracciamo meglio è, ci è facile cadere nella sovrabbondanza di dati che finiscono poi per creare più confusione e rendere più complicata la diagnostica di un problema. Per questo motivo, ogni tracciamento che facciamo viene accompagnato da un livello e per ognuno di essi disponiamo di un extension method per facilitarci il loro utilizzo. Nell'Esempio 13.12 possiamo vederli all'opera, in ordine d'importanza.

Esempio 13.12

```
_logger.LogTrace("Linea dettagliata con potenziali informazioni sensibili");

_logger.LogDebug(1001, "Informazioni di debug, utili per diagnostica");

_logger.LogInformation("Dati generici e descrittivi");

_logger.LogWarning("Avviso non bloccante");

_logger.LogError(5001, exception, "Si è verificato un errore, bloccante");

_logger.LogCritical(exception, "Errore catastrofico, perdita di dati");
```

Tutti gli extension method dell'Esempio 13.12 dispongono di vari overload che permettono di specificare template, parametri e anche l'oggetto `Exception`, utile soprattutto quando utilizziamo le funzioni `LogError` e `LogCritical`. Facoltativamente, il primo parametro di ogni funzione accetta un identificativo, un intero che ci permette di identificare il tracciamento più precisamente di una stringa. Il livello, nell'Esempio 13.12, utilizzato in ordine crescente, ci permette di scremare il dettaglio di informazioni da mostrare o da persistere. È di fondamentale importanza tracciare le informazioni utilizzando il livello più appropriato e non cadere nella pigrizia che spesso porta a utilizzare uniformemente il livello `Information`.

I livelli hanno un'importanza che non è solo formale ma anche pratica. Ogni provider può decidere autonomamente quali livelli onorare, dando un livello minimo. Se, per esempio, il debug ha un livello minimo su `Warning`, vengono memorizzati eventi solo di livello uguale o superiore, quindi: `Warning`, `Error` e `Critical`. Il provider dedicato alla console, produce l'output della Figura 13.5 con il suo comportamento predefinito. Quando utilizziamo IIS Express, esso è visibile attraverso una finestra apposita di Visual Studio.

```
Output
Show output from: ASP.NET Core Web Server
Capitolo13> info: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Index with my and 2
Capitolo13> dbug: Capitolo13.Controllers.LoggingController[1001]
Capitolo13>       Informazioni di debug, utili per diagnostica
Capitolo13> info: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Dati generici e descrittivi
Capitolo13> warn: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Avviso non bloccante
Capitolo13> fail: Capitolo13.Controllers.LoggingController[5001]
Capitolo13>       Si è verificato un errore, bloccante
Capitolo13> System.Exception: test
Capitolo13> crit: Capitolo13.Controllers.LoggingController[0]
Capitolo13>       Errore catastrofico, perdita di dati
Capitolo13> System.Exception: test
Capitolo13> info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
Capitolo13>       Executing HttpStatusCodeResult, setting HTTP status code 200
```

Figura 13.5 – Output in console attraverso Visual Studio.

Nella figura possiamo vedere una formattazione del testo che mostra il livello abbreviato, il messaggio formattato e, tra parentesi quadre, i nostri identificativi 1001 e 5001 utilizzati e, laddove non specificati, assunti con valore a 0. La categoria, dedotta dal tipo T di `ILogger` richiesto nell'Esempio 13.1, ricopre un ruolo importante, perché rappresenta un ulteriore strumento per differenziare e ricercare le informazioni tracciate. Nella Figura 13.5, all'ultima riga, possiamo vedere che non siamo gli unici a sfruttare questo strumento. L'intero ASP.NET Core è scritto affinché anch'esso tracci, con categorie sotto il namespace `Microsoft`, livelli e identificativi differenti.

Con il tracciamento abbiamo anche la possibilità di circoscrivere in uno scope più messaggi all'interno di un unico contesto, anch'esso identificato dal messaggio e da eventuali parametri. Nell'Esempio 13.13 vediamo come aprire uno scope e come chiuderlo grazie al pattern dispose.

Esempio 13.13

```
// Inizio di un nuovo scope con parametri
using (_logger.BeginScope("Nuovo scope {mode}", mode))
{
   _logger.LogInformation("Dati generici e descrittivi");
   _logger.LogWarning("Avviso non bloccante");
}
```

Gli scope non sono normalmente visualizzati in console, perché amplificano la quantità di informazioni mostrate, ma è possibile agire su essa tramite l'opzione `IncludeScopes` in fase di configurazione del provider.

Configurare il logging

Quando creiamo un nuovo progetto ASP.NET Core, godiamo automaticamente di una serie di configurazioni implicite, che ci permettono di partire immediatamente con lo sviluppo. Il metodo `CreateDefaultBuilder` che troviamo nel `Program.cs` effettua una serie di configurazioni dedicate al logging, che possiamo trovare ricostruite nell'Esempio 13.14.

Esempio 13.14

```
webhost.
   .ConfigureLogging((h, l) =>
   {
      l.AddConfiguration(h.Configuration.GetSection("Logging"));
      l.AddConsole();
      l.AddDebug();
})
```

La funzione di configurazione `ConfigureLogging` permette di passare un delegato che accetta il contesto di configurazione (parametro h) e un `ILoggingBuilder` (parametro l) che, nello stile di .NET Core, permette di aggiungere uno o più provider. La chiamata a `AddConfiguration` aggancia la sezione `Logging` della configurazione, per consentirci di personalizzare alcuni aspetti di logging senza dover ricompilare il nostro codice. Le chiamate `AddConsole` e `AddDebug` aggiungono rispettivamente i provider che scrivono in console e nella finestra di debug di Visual Studio.

In uno scenario reale, quel codice è implicitamente presente e a noi resta solo il compito di una personalizzazione di `Program.cs`, sfruttando, per esempio, altri provider che troviamo implementati.

Esempio 13.15

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
      .ConfigureLogging(l =>
      {
         // Aggiungo i provider
         l.AddEventSourceLogger();
         l.AddTraceSource("mySwitch");
         l.AddProvider(myLoggerProviderInstance);
      // Imposto il livello minimo
      l.SetMinimumLevel(LogLevel.Debug);
      })
      .UseStartup<Startup>()
.Build();
```

L'extension method `AddEventSourceLogger` configura l'utilizzo del registro di sistema per la memorizzazione dei propri tracciamenti, mentre `AddTraceSource` configura la scrittura su `System.Diagnostics.Trace`, il precedente sistema di tracciamento nato con il .NET Framework. Infine, la chiamata a `AddProvider` permette di aggiungere un'istanza personalizzata di provider, anche se nella maggior parte dei casi sono sufficienti gli extension method di configurazione. Cercando all'interno di NuGet, è facile trovare altri provider che, una volta installati, vi mettono a disposizione un nuovo extension method con il prefisso `Add` da chiamare.

Nell'Esempio 13.15 troviamo, infine, una chiamata a `SetMinimumLevel`, con il quale possiamo forzare il livello minimo che ogni provider riceve. Così facendo, il motore va automaticamente a escludere tracciamenti di livello inferiore, in maniera del tutto trasparente al provider.

Non solo: in alternativa all'istruzione via codice, possiamo sfruttare l'aggancio fatto alla configurazione nell'Esempio 13.14, per gestire questo comportamento direttamente dal file `appsettings.json` o tramite altri provider, a seconda di come il sistema delle opzioni è stato impostato. Quando creiamo un progetto direttamente con Visual Studio, troviamo due file: `appsettings.json` e `appsettings.Development.json`. Il secondo, contenente la configurazione letta per le fasi di sviluppo, contiene il JSON dell'Esempio 13.16.

Esempio 13.16

```
{
   "Logging": {
      "IncludeScopes": false,
      "LogLevel": {
         "Default": "Debug",
         "System": "Information",
         "Microsoft": "Information"
      }
   }
}
```

La sezione `Logging` contiene un nodo `LogLevel` e una chiave `Default` con la quale viene configurato il livello minimo da adottare, allo stesso modo di quanto fatto via codice nell'Esempio 13.15. Non solo: per ogni categoria o prefisso di categoria, vengono impostati livelli diversi. Quello che otteniamo è la visualizzazione in console di tutti gli eventi per qualsiasi livello, mentre per quelli generati da `Microsoft`, il livello dev'essere uguale o superiore a `Information`. La Figura 13.5, che abbiamo già visto, lo dimostra e, se necessario, possiamo abilitare il livello `Debug` o `Trace` anche sui tracciamenti di ASP.NET Core. Possiamo inoltre aggiungere una o più voci dedicate alle nostre categorie, circoscrivendole a `Capitolo13` o `Capitolo13.Controllers`.

Le possibilità non sono finite, perché possiamo configurare il livello da adottare anche per uno specifico provider o per una specifica categoria di un certo provider, come mostrato nell'Esempio 13.17.

Esempio 13.17

```
{
   "Logging": {
      "IncludeScopes": false,
      "LogLevel": {
         "Default": "Debug",
         "System": "Information",
         "Microsoft": "Information",
         "Capitolo13": "Trace"
      },
      "Console": {
         "LogLevel": {
            "Microsoft.AspNetCore.Mvc.Razor.Razor": "Debug",
            "Microsoft": "Error",
            "Default": "Information"
         }
      }
   }
}
```

Una volta identificato il provider sul quale vogliamo intervenire, la sintassi da usare è la medesima vista in precedenza: prefisso o nome completo della categoria, e `Default` per tutte le altre categorie. Dobbiamo sottolineare l'ordine di precedenza delle impostazioni fatte, poiché vince prima di tutto la categoria più specifica, mentre a parità di categoria vince l'ultima definita. In entrambi i casi, è indifferente se la regola è stata impostata a livello di provider o in modo generico, come è stato fatto nell'Esempio 13.16. Interessante, infine, è sapere che le modifiche alla configurazione vengono applicate immediatamente senza necessità di riavviare l'host.

Quanto è stato fatto da configurazione è possibile farlo anche da codice, se per caso volessimo godere della massima libertà di espressione, come viene mostrato nell'Esempio 13.18.

Esempio 13.18

```
.ConfigureLogging(l =>
{
   l.AddFilter((provider, category, logLevel) =>
   {
      // Solo i nostri eventi informativi
      if (logLevel == LogLevel.Information && category.StartsWith("Capitolo13"))
      {
         return true;
      }
      return false;
   });
})
```

Il metodo `AddFilter` dispone di molti overload per scegliere quali parametri o a quale categoria ci rivolgiamo ma, nell'Esempio 13.18, utilizziamo quello più versatile perché fornisce tutte le informazioni necessarie alla nostra logica e ci permette di indicare se tracciare (restituendo true) oppure no. È importante tenere presente che questo filtro viene chiamato solo se non abbiamo specificato un livello predefinito nella configurazione e solo per le categorie orfane di livello minimo. Visti tutti gli elementi che compongono il logging, non ci resta che vedere come implementare un provider personalizzato.

Un provider per il logging personalizzato

L'architettura estremamente modulare di ASP.NET Core ci permette anche nel logging di fornire un provider personalizzato, al pari degli altri già implementati. `Console` e `Debug` sono spesso insufficienti e può rendersi necessario l'invio di informazioni a sistemi di tracciamento esterni, database, sistemi di comunicazione o anche al semplice invio di un'e-mail al verificarsi di un errore. Un provider è rappresentato dall'interfaccia `ILoggerProvider` che occorre registrare nel container, come tutte le altre dipendenze. In linea con il

pattern di configurazione dell'intero .NET Core, possiamo quindi creare un extension method per l'aggiunta di un nostro ipotetico provider che manda l'e-mail.

Esempio 13.19

```
namespace Microsoft.Extensions.Logging
{
   public static class EmailLoggerProviderExtensions
   {
      public static ILoggingBuilder AddEmail(this ILoggingBuilder builder)
      {
         builder.Services.AddSingleton<ILoggerProvider, EmailLoggerProvider>();
         return builder;
      }
   }
}
```

L'uso del namespace dedicato al logging permette in fase di configurazione, come nell'Esempio 13.15, di chiamare `AddEmail` con il supporto dell'IntelliSense e senza aggiungere ulteriori namespace. Un provider non è altro che un factory che genera istanze diverse di `ILogger` per la categoria indicata, come viene mostrato nell'Esempio 13.20.

Esempio 13.20

```
[ProviderAlias("Email")]
public class EmailLoggerProvider : ILoggerProvider
{
   private readonly ConcurrentDictionary<string, EmailLogger> _loggers = new
ConcurrentDictionary<string, EmailLogger>();

   public ILogger CreateLogger(string categoryName)
   {
      // Caching dei logger
      return _loggers.GetOrAdd(categoryName, c => new EmailLogger(c));
   }
   public void Dispose()
   {
      _loggers.Clear();
   }
}
```

Poiché le categorie sono un numero limitato e definito, è buona norma effettuare un caching delle istanze, per non dover istanziare ogni volta l'oggetto, dato che il logging è uno strumento che viene spesso invocato. L'uso del `ConcurrentDictionary` garantisce la memorizzazione e la creazione di istanze al riparo da problemi di race condition tra i thread. L'uso dell'attributo `ProviderAlias` permette di specificare il nome con il quale fare riferimento all'interno della configurazione, come nell'Esempio 13.17, per indicare che vogliamo un livello minimo di tipo `Error` per questo specifico provider.

Il cuore del nostro provider è l'implementazione di `EmailLogger`, il quale implementa `ILogger` e principalmente il metodo `Log`, che a sua volta ha il compito di memorizzare o, come nel nostro caso, di inviare l'e-mail.

Esempio 13.21

```
public class EmailLogger : ILogger
{
   private string _category;
   public EmailLogger(string category)
   {
      _category = category;
   }
   public void Log<TState>(
      LogLevel logLevel,
      EventId eventId,
      TState state,
      Exception exception,
      Func<TState, Exception, string> formatter)
   {
      // Accedo allo stato come coppia chiave/valore
      var values = state as IEnumerable<KeyValuePair<string, object>>;
      // Serializzo lo stato
      string json = JsonConvert.SerializeObject(values);
      // Ottengo il messaggio
```

```
        string message = formatter(state, exception);
        // Invio dell'e-mail...
    }
    public bool IsEnabled(LogLevel logLevel) => true;
    public IDisposable BeginScope<TState>(TState state) => null;
}
```

Possiamo vedere che il metodo `Log` riceve tutte le informazioni: il livello, l'identificativo, lo stato, l'eventuale eccezione e una funzione per ottenere il messaggio formattato. Per una questione di prestazioni, infatti, spetta a noi decidere se formattare o no il messaggio per rendere più leggera l'operazione di tracciamento. Lo stato può essere di qualsiasi tipo anche se tramite gli extension method dell'Esempio 13.13 è sempre un oggetto che dà una lista di coppie chiave/valore. Nell'Esempio 13.21 non facciamo altro che serializzare in JSON lo stato, per mandarlo ipoteticamente come allegato all'email. L'implementazione di quest'ultima parte è omessa perché esula dagli argomenti trattati in questo capitolo, ma è bene tenere in considerazione che il metodo `Log` potrebbe essere richiamato più volte e dovrebbe quindi essere il più veloce possibile. L'invio dell'e-mail o la memorizzazione su database sono operazioni che sarebbe opportuno fare in asincrono o con un meccanismo di buffer che accumula un po' di messaggi e li invia tutti insieme quando il buffer è pieno oppure ripetendosi a intervalli regolari.

Le potenzialità di un provider sono quindi infinite e, una volta poste le basi, possiamo integrarci con tutti i sistemi, come per esempio Microsoft Azure.

L'integrazione con Azure App Service

Una particolarità che contraddistingue lo sviluppo con le tecnologie Microsoft è sicuramente costituita dalla completezza nella soluzione che mette a disposizione. Oltre al framework web, cioè ASP.NET Core e all'ambiente di sviluppo con Visual Studio, disponiamo anche della possibilità di ospitare i nostri applicativi sulla piattaforma cloud di nome Microsoft Azure. Nello specifico, gli App Service sono un servizio completamente gestito, scalabile e altamente affidabile, dove possiamo distribuire i nostri applicativi web esponendoli con Windows Server e IIS o con Linux e nginx, praticamente con la totalità dei linguaggi e delle tecnologie, tra cui ASP.NET Core. La facilità di distribuzione, la potenza e la presenza anche di piani gratuiti rendono molto appetibile l'ambiente, quantomeno per lo sviluppo.

Quando utilizziamo questo servizio, godiamo anche di una gestione del logging e un aiuto nella diagnostica. Esiste, infatti, la possibilità di scrivere del log su file system oppure su blob, il servizio distribuito di persistenza dei file, fornito sempre da Microsoft Azure. È quindi naturale sfruttare il sistema per usufruire di un salvataggio affidabile e consultabile senza accesso al server, dato che i servizi PaaS, come quello di App Service, non lo consentono.

L'integrazione tra ASP.NET Core e Microsoft Azure è così forte che nella fase di configurazione, vista nell'Esempio 13.15, troviamo anche un extension method di nome `AddAzureWebAppDiagnostics`. Non solo: chiamarlo è perfino superfluo, perché il provider si installa automaticamente qualora dovessimo eseguire il nostro applicativo sul cloud. L'unica richiesta da fare è quella di pubblicare l'applicativo su un App Service, anche direttamente da Visual Studio, e di abilitare la diagnostica. Questo libro non è una guida sull'utilizzo della piattaforma, ma vogliamo quantomeno mostrare come integrarci nella diagnostica. Non appena entrati nel portale e nella sezione della nostra app, è sufficiente aprire la sezione "Log di diagnostica". Il pannello, visibile nella Figura 13.6, permette di abilitare il logging su file system o su blob.

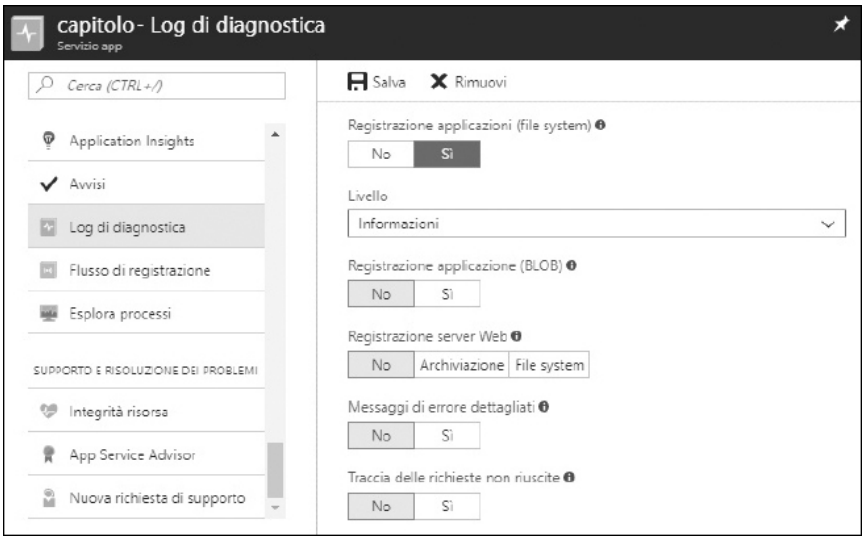


Figura 13.6 – Abilitazione del logging sul portale di Microsoft Azure.

Possiamo cambiare il livello minimo da registrare in qualsiasi momento e, successivamente, possiamo recuperare il log sul blob selezionato o tramite FTP, navigando nella cartella apposita indicata più sotto nella sezione della Figura 13.6.

Non solo: sempre dal portale possiamo aprire la sezione “Flusso di registrazione”, che permette di attivare una visualizzazione live del logging fornito dal provider. Possiamo vedere nella Figura 13.7 come riusciamo a visualizzare le stesse informazioni ottenute nella Figura 13.5, questa volta attraverso un’interfaccia web.

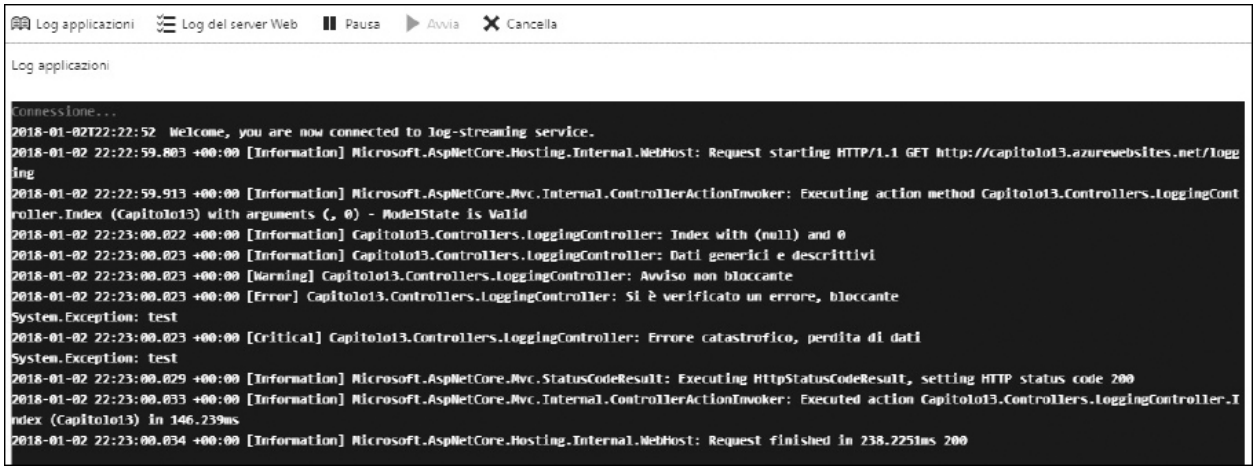


Figura 13.7 – Flusso di visualizzazione live del log sul portale di Microsoft Azure.

L’integrazione, a conti fatti, è talmente banale che diventa pressoché automatico sfruttare questo provider se stiamo distribuendo il nostro applicativo su Microsoft Azure.

Conclusioni

In questo capitolo abbiamo affrontato tutto ciò che ASP.NET Core mette a disposizione affinché il nostro applicativo possa supportare eventuali errori e consenta una facile diagnostica.

Middleware dedicati ci permettono da un lato di avere un supporto durante lo sviluppo, dall’altro di fornire pagine amichevoli per rendere seria l’app. A questo scopo viene in aiuto anche la personalizzazione delle pagine di stato, possibile anche in questo caso tramite view per MVC o per Razor Page. Strumenti dedicati al logging ci permettono di tracciare tutto ciò che avviene nella nostra app, permettendoci di organizzare le informazioni per livello, categoria, id e stati che possono essere memorizzati o visualizzati in maniera strutturata. Un’architettura a provider ci permette di sfruttare più componenti in grado di gestire contemporaneamente informazioni di tipo diverso e, con poco sforzo, possiamo realizzare un oggetto personalizzato che possa integrarsi con qualsiasi sistema esterno.

Sul tema della personalizzazione vogliamo proseguire anche nel prossimo capitolo, cercando di capire meglio alcuni fra gli aspetti principali che muovono il motore di ASP.NET Core.

14
Estendere ASP.NET Core

Quanto abbiamo affrontato nei capitoli precedenti è sufficiente per cominciare a sviluppare applicazioni con ASP.NET Core, mostrare delle form, offrire API e accedere al database. Ma se vogliamo trarre il massimo dal framework, è necessario approfondire alcune caratteristiche più avanzate che si possono dimostrare fondamentali qualora i nostri progetti crescessero. Diventa sempre più probabile, infatti, poter sfruttare le caratteristiche di dinamicità offerte da ASP.NET Core per la realizzazione di moduli e logiche che possano essere facilmente riutilizzati, nel progetto stesso o in librerie separate, al fine di essere sfruttati più volte.

Nel framework disponiamo di questa possibilità, perciò in questo capitolo partiremo dall'estensibilità offerta dal punto di vista dell'hosting, approfondendo alcuni meccanismi, quindi vedremo come sfruttare la pipeline dei middleware per realizzare componenti, come realizzare librerie contenenti view riutilizzabili, utili anche per una migliore organizzazione dei progetti, fino a entrare nei dettagli di funzionamento di MVC e realizzare filtri personalizzati che lavorino sulla pipeline di esecuzione.

Personalizzare l'host di esecuzione

Nel Capitolo 3 abbiamo affrontato le caratteristiche principali dell'host di un'applicazione ASP.NET Core e come questo venga creato nel `Program.cs` attraverso la chiamata a `WebHost.CreateDefaultBuilder`. L'oggetto restituito offre un'interfaccia alla quale vengono agganciati vari comportamenti predefiniti, che vanno a configurare il motore di dependency injection, a impostare alcuni servizi e ad attivare la classe `Startup`. In quest'ultima, fondamentalmente andiamo a registrare i servizi e i middleware da utilizzare attraverso i più disparati extension method forniti dalle librerie built-in o di terze parti.

Alcuni dei comportamenti predefiniti, però, non sono inseriti nell'implementazione base di ASP.NET Core, ma sono iniettati in base all'ambiente in cui vengono eseguiti. Abbiamo visto nel Capitolo 13, per esempio, che semplicemente caricando l'applicativo di un ambiente virtualizzato di Microsoft Azure, possiamo utilizzare un provider di logging integrato. Questo è possibile grazie al motore e alla facoltà di iniettare delle dipendenze allo startup dell'applicativo.

Iniettare l'utilizzo di una libreria

Sebbene il pacchetto NuGet `Microsoft.AspNetCore.App` contenga moltissimi assembly, ciò non significa che questi vengano tutti usati, ma solamente che questi sono messi a disposizione per la fase di compilazione. Chiamando poi i rispettivi extension method, come `AddMvc` o `UseMvc`, andiamo a personalizzare l'ambiente host, ma per la registrazione automatica di alcune librerie il motore riserva una via alternativa. Quando avviamo il debug da Visual Studio, per esempio, otteniamo automaticamente la registrazione dell'integrazione con IIS, normalmente non attiva. Questo è possibile grazie a una variabile d'ambiente, di nome `ASPNETCORE_HOSTINGSTARTUPASSEMBLIES` impostata da Visual Studio e contenente una lista di assembly da caricare dinamicamente; Il discovery non è automatico, questo per mantenere alte le prestazioni di avvio del nostro applicativo. Diversamente il motore dovrebbe caricare tutte le dll, per analizzare se sono presenti tipi che soddisfano i requisiti. Questo meccanismo può essere di conseguenza sfruttato anche da una nostra libreria, nel caso in cui non volessimo configurarla manualmente da codice, ma attivarla tramite variabile d'ambiente.

Il primo passo da fare è creare una libreria, referenziare il pacchetto `Microsoft.AspNetCore.Hosting.Abstractions` e implementare con una nostra classe l'interfaccia `IHostingStartup`. L'unico metodo da implementare ci permette di lavorare direttamente con `IWebHostBuilder`, perciò di avere il pieno controllo dell'host. L'istruzione più comune da usare è quella che ci permette di configurare servizi utili alla libreria, come viene mostrato nell'Esempio 14.1.

Esempio 14.1

```
public class MyHostingStartup : IHostingStartup
{
   public void Configure(IWebHostBuilder builder)
   {
      builder.ConfigureServices(s =>
      {
         // Configurazione di un servizio della libreria
         s.AddTransient<MyService>();
      });
   }
}
```

La tecnica è del tutto simile a quanto già facciamo nella classe `Startup`, ma definendo il tutto in una libreria esterna. Occorre successivamente aggiungere, fuori dai namespace, un attributo di assembly, per indicare al motore quale tipo implementa l'interfaccia, come è visibile nell'Esempio 14.2.

Esempio 14.2

```
[assembly: HostingStartup(typeof(MyHostingStartup))]
```

Tramite attributo, anche in questo caso consentiamo una ricerca più rapida, per evitare che il motore si scorra tutte le classi.

Il passaggio successivo richiede la distribuzione della libreria, referenziandola direttamente nel progetto o distribuendola insieme alle altre dipendenze. Non resta altro che valorizzare la variabile d'ambiente prima citata a seconda delle modalità di avvio dell'host. Ai fini di sviluppo, tramite Visual Studio possiamo valorizzare la variabile insieme a ASPNETCORE_ENVIRONMENT come abbiamo già visto nella Figura 3.7. In alternativa, possiamo chiamare la funzione `UseSetting` di `IWebHostBuilder`, come mostrato nell'Esempio 14.3.

Esempio 14.3

```
public static IWebHostBuilder BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
         .UseStartup<Startup>()
         .UseSetting(WebHostDefaults.HostingStartupAssembliesKey, "Capitolo14.Library");
```

L'uso della variabile d'ambiente rimane comunque la tecnica migliore perché ci permette dall'esterno di influenzare i componenti da registrare, facoltà fornita dagli ambienti cloud e da motori di virtualizzazioni come Docker, che verrà affrontato nel Capitolo 22. Con questa tecnica possiamo dunque creare un insieme di servizi in una libreria e registrarli dinamicamente, ma possiamo andare oltre e configurare l'applicazione separatamente.

Configurazione separata dell'app

Nel file di startup, oltre al metodo `ConfigureServices`, dove configuriamo i servizi, disponiamo del metodo `Configure` che, ricevendo un oggetto di tipo `IApplicationBuilder` ci permette di impostare principalmente i middleware da usare, fondamentali per processare le richieste. Con l'Esempio 14.1, purtroppo, non abbiamo questa facoltà, perché ci è concesso solo di configurare i servizi. Viene in aiuto l'interfaccia `IStartupFilter`, che ci permette di eseguire questo ulteriore passaggio. È sufficiente quindi implementare l'interfaccia in una nuova classe e registrarla tra i servizi impostati nell'Esempio 14.1.

L'ambiente di host all'avvio non fa altro che sfogliare tutti i servizi registrati di tipo `IStartupFilter` e li invoca nell'ordine in cui sono stati registrati. Nella loro implementazione, come è visibile nell'Esempio 14.4, dobbiamo restituire un delegato che effettui le operazioni di configurazione di `IApplicationBuilder`.

Esempio 14.4

```
public class MyStartupFilter : IStartupFilter
{
   public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
   {
      return builder =>
      {
         // Chiamo le altre configurazioni
         next(builder);
         // Aggiungo in coda i miei middleware
         builder.UseMvc();
      };
   }
}
```

La funzione deve restituire un delegato in grado di configurare l'oggetto, ma allo stesso tempo riceve un delegato che invoca le altre implementazioni di `IStartupFilter`, compresa quella presente nello `Startup.cs`. Questo ci permette di intervenire prima o dopo di esse, per dare ordine alla priorità dei middleware.

Tra i servizi utili all'estensione dell'ambiente di host vi è inoltre la possibilità di agganciare servizi in background.

Avviare servizi in background

Middleware, controller e view, tra i principali, sono oggetti che vengono tutti chiamati in causa a fronte di una richiesta da parte dell'utente, perché è questo il compito principale del nostro applicativo. Dato però che esso non è altro che una console sempre attiva, è legittimo pensare che essa possa svolgere altre operazioni, sfruttando thread diversi. Sebbene questo si possa realizzare modificando il `Program.cs`, in .NET Core esiste un approccio standard per realizzare servizi che lavorino in background, indipendentemente dalla parte ASP.NET.

È sufficiente, infatti, implementare l'interfaccia `IHostedService` e registrarla come di consueto nel motore delle dipendenze, nello `Startup` oppure in `IHostingStartup`. Essa dispone di due funzioni `StartAsync` e `StopAsync`, invocate rispettivamente all'avvio, non appena la fase di preparazione dell'applicazione si è conclusa, e all'arresto, non appena viene mandato il segnale di chiusura. Il servizio è quindi ideale per compiere, per esempio, operazioni di setup oppure per avviare un timer che svolga attività di controllo. Nell'Esempio 14.5 possiamo vedere un'implementazione base che prepara proprio un timer, il cui scopo è di controllare un database di utenze e mandare degli avvisi.

Esempio 14.5

```
public class PingCustomerHostedService : IHostedService
```

```
{
   private System.Timers.Timer _pingTimer;
   public Task StartAsync(CancellationToken cancellationToken)
   {
      // Ogni 10 minuti
      int interval = 1000 * 60 * 10;
      _pingTimer = new System.Timers.Timer(interval);
      _pingTimer.Elapsed += OnPingTimer;
      _pingTimer.Start();
      return Task.CompletedTask;
   }
   public Task StopAsync(CancellationToken cancellationToken)
   {
      _pingTimer?.Stop();
      return Task.CompletedTask;
   }
}
```

L'ambito in cui lavorano le funzioni `StartAsync` e `StopAsync` è disconnesso dal motore web che si mette in ascolto ed è indipendente dalla durata delle attività che eseguiamo all'avvio che, per influenzare il meno possibile altri servizi, dovrebbero eseguire un'attività asincrona breve e, se necessario, lavorare su altri thread per attività lunghe. Questa separazione va tenuta in considerazione nel momento in cui facciamo utilizzo della dependency injection, perché tutto ciò che possiamo ottenere nel costruttore della nostra classe dev'essere stato registrato con ciclo di vita transient o singleton. Ciò che è stato registrato in modalità scope non può essere richiesto, perché non esiste alcuno scope e solitamente questo è circoscritto dal ciclo di vita di una richiesta dell'utente, mancante in questo ambito.

Dobbiamo di conseguenza chiedere nel costruttore un'istanza di `IServiceProvider` e provvedere manualmente alla creazione di uno scope. Tramite quest'ultimo, come è mostrato nell'Esempio 14.6, possiamo poi ottenere un'istanza del servizio che necessita di uno scope, come, per esempio, è un contesto di Entity Framework.

Esempio 14.6

```
public class PingCustomerHostedService : IHostedService, IDisposable
{
   private readonly IServiceProvider _serviceProvider;
   public PingCustomerHostedService(IServiceProvider serviceProvider)
   {
      _serviceProvider = serviceProvider;
   }
   // ... omessi StartAsync e StopAsync
   private void OnPingTimer(object sender, ElapsedEventArgs e)
   {
      // Creazione esplicita dello scope
      using (IServiceScope scope = _serviceProvider.CreateScope())
      {
         var context = scope.ServiceProvider
            .GetRequiredService<CustomersContext>();

         // Interrogazione clienti
      }
   }
   public void Dispose()
   {
      _pingTimer?.Dispose();
   }
}
```

Nel codice possiamo vedere la creazione esplicita dello scope, il recupero del servizio interessato e la conseguente distruzione del primo, non più necessario. Nell'esempio possiamo inoltre notare l'implementazione dell'interfaccia `IDisposable`, che possiamo sfruttare per chiudere gli oggetti utilizzati internamente nel momento in cui il container della dependency injection distruggerà il nostro oggetto. Per quanto riguarda la funzione `StopAsync`, va sottolineato che l'operazione asincrona, per garantire la chiusura dell'applicativo, dispone di un tempo massimo di esecuzione di cinque secondi, oltre il quale il processo termina comunque. Se proprio è necessario, possiamo aumentare questo limite nella configurazione dell'host, come viene mostrato nell'Esempio 14.7.

Esempio 14.7

```
public static IWebHostBuilder BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
```

```
        .UseStartup<Startup>()
 .UseShutdownTimeout(TimeSpan.FromSeconds(10));
```

Quanto visto ci permette di organizzare il nostro codice in librerie e di agganciarci agevolmente all'ambiente di host, ma questo approccio è utile anche per strutturare controller e view del pattern MVC.

Creare librerie con view Razor

La strutturazione di controller e view all'interno del progetto principale del nostro applicativo è una convenzione che ci permette immediatamente di partire con lo sviluppo delle pagine. I controller sono, di fatto, delle normali classi e possono essere posizionati in altre librerie senza nessuna particolare configurazione, poiché il motore di ASP.NET MVC cerca in tutti gli assembly referenziati. Questo è particolarmente utile quando i progetti diventano grossi e le aree non sono sufficienti, oppure perché vogliamo poter riutilizzare una parte delle pagine su altri progetti.

Purtroppo, non disponiamo di questa possibilità sulle view, perché esse necessitano di essere compilate e il motore deve conoscere dove sono posizionate e a quale percorso rispondono. Esiste però la possibilità di creare una particolare libreria di tipo Razor Class Library, attraverso la seconda finestra di dialogo che si presenta quando scegliamo di creare o aggiungere una nuova ASP.NET Core Web Application, che abbiamo già visto nella Figura 2.6. Non appena creata, la libreria si presenta con un csproj simile al seguente.

Esempio 14.8

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
   <PropertyGroup>
      <TargetFramework>netcoreapp2.1</TargetFramework>
   </PropertyGroup>
   <ItemGroup>
      <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="2.1.0" />
   </ItemGroup>
</Project>
```

L'uso di un SDK dedicato, già incluso in Visual Studio, evidenziato nell'Esempio 14.8, ci consente di inserire view e di compilarle. Successivamente, la libreria potrà essere referenziata nell'applicativo che ne necessita e, senza effettuare altre operazioni, potremo utilizzare le view Razor contenute al suo interno. La realizzazione di una libreria con controller, model, view e partial view non differisce da quanto facciamo già sull'applicativo principale. Posizioniamo i rispettivi file nelle relative directory rispettando le convenzioni, a seconda dei nomi dei controller, delle action e delle aree. Nella Figura 14.1 possiamo vedere un esempio di strutturazione che rispetta le convenzioni.

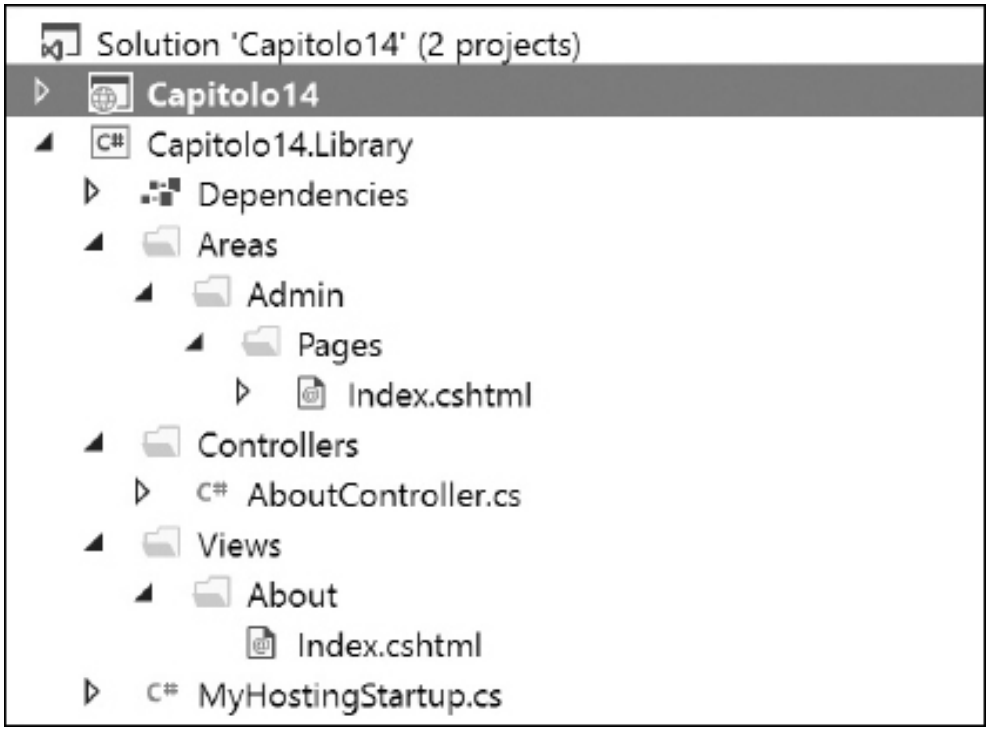


Figura 14.1 – Strutturazione dei file in una Razor Class Library in Visual Studio.

La figura mostra la presenza di un `AboutController` e della relativa view `Index.cshtml`. Dispone inoltre di un'area di nome `Admin` contenente una Razor Page, quindi senza controller, di nome `Index.cshtml`. Le due pagine vengono unite a quanto presente sull'applicativo principale e sono raggiungibili rispettivamente all'indirizzo `/About` (Controller=About, Action=Index, Area=null) e `/Admin` (Page=Index, Area=Admin).

> *Il risultato della compilazione di un Razor Class Library, così come dell'applicativo principale, è sempre composto da due file: nomeLibreria.dll e nomeLibreria.Views.dll. La seconda contiene le view pronte per essere eseguite il più velocemente possibile.*

L'aspetto interessante dell'unione della struttura dei file è il fatto che viene permessa anche la sovrascrittura parziale delle view definite nelle librerie. Ponendo di voler personalizzare la view `About/Index.cshtml`, è sufficiente mettere il medesimo file nello stesso percorso dell'applicativo, lasciando intatta la libreria, e otterremo così la sovrascrittura a runtime di una view, mantenendo inalterate le altre. Questa tecnica è particolarmente utile in `One Identity`, una libreria che affronteremo nel Capitolo 17, la quale porta con sé delle interfacce standard ma che, se necessario, possiamo personalizzare.

Caricare a runtime parti dell'applicazione

Lo sviluppo di librerie è una pratica molto comune e consigliata, perché ci permette di organizzare meglio il codice, di lavorare più facilmente in team e di riutilizzare funzionalità in più applicativi. Non importa se sono delle normali Class Library o delle Razor Class Library, perché in entrambi i casi le dobbiamo referenziare all'interno dell'applicazione principale. Per una questione di ottimizzazione, quando il motore di ASP.NET MVC parte, ispeziona le referenze dell'applicativo stesso e non scorre i file fisicamente presenti nella cartella. Questo meccanismo è gestito dall'`ApplicationPartManager`, un oggetto che raccoglie gli Application Part, i quali, a loro volta, forniscono le feature dell'applicativo, cioè tutti i controller, metadati, tag helper e view disponibili. È grazie a essi che il nostro applicativo MVC è in grado di trovare ed eseguire tutti questi componenti.

L'aspetto interessante è che possiamo intervenire e manipolare quella collezione per decidere di rimuovere o aggiungere Application Part per raggiungere i nostri scopi. Poniamo di voler realizzare un applicativo che fornisca delle funzioni base, ma grazie a una cartella di nome `plugin` permetta di inserire le dll che ne arricchiscono le funzionalità dello stesso. Procediamo quindi a creare una Razor Class Library ma, diversamente da quanto è stato fatto nel paragrafo precedente, non la referenziamo nell'applicativo. Per un più facile sviluppo, andiamo nelle proprietà della libreria e, nella sezione `Build Events`, inseriamo il seguente script DOS.

Esempio 14.9

```
set d="$(SolutionDir)bin\$(ConfigurationName)\netcoreapp2.1\plugin"
if not exist %d% mkdir %d%
copy "$(TargetDir)*.dll" %d%
```

Lo script dell'Esempio 14.9 copia tutte le dll generate dalla compilazione della libreria nella directory `plugin` dell'applicativo principale, evitandoci di dover eseguire manualmente questa operazione. Alla luce di quanto abbiamo detto in precedenza, però, questo non è sufficiente, perché ASP.NET MVC non va a caricare questo assembly, ma siamo noi a doverlo fare esplicitamente.

Nello `Startup.cs` riprendiamo il metodo `ConfigureServices` andando a configurare l'`ApplicationPartManager`, dapprima cercando le dll presenti nella cartella, successivamente andandole a caricare, come viene mostrato nell'Esempio 14.10.

Esempio 14.10

```
services.AddMvc()
   .SetCompatibilityVersion(CompatibilityVersion.Version_2_1)
   .ConfigureApplicationPartManager(p =>
   {
      // Ottengo la directory dove sono presenti i plugin
      string dir = Path.GetDirectoryName(typeof(Startup).GetTypeInfo().Assembly. Location);
      dir = Path.Combine(dir, «plugin»);
      if (!Directory.Exists(dir)) return;
      // Ciclo tutte le dll
      foreach (string file in Directory.GetFiles(dir, "*.dll"))
      {
         // Carico l'assembly
         Assembly assembly = Assembly.LoadFile(file);
         // Aggiungo l'application part per i componenti
         p.ApplicationParts.Add(new AssemblyPart(assembly));
         // Aggiungo l'application part per le view
         p.ApplicationParts.Add(new CompiledRazorAssemblyPart(assembly));
      }
});
```

Il codice, commentato nelle sue parti, scorre tutte le dll, chiama il metodo statico `Assembly.LoadFile` per caricare dinamicamente l'assembly e popola la collezione `ApplicationParts`. In essa sono già presenti tutte le Application Part trovate dal motore allo startup e a essa aggiungiamo due tipologie, che vanno entrambe a cercare nell'assembly caricato:

- `AssemblyPart`: cerca i controller e i tag helper contenuti nell'assembly.
- `CompiledRazorAssemblyPart`: cerca le view compilate all'interno dell'assembly.

Poiché nella directory plugin ci potrebbero essere assembly contenenti diverse tipologie di componenti che però ignoriamo, inseriamo entrambe le tipologie. Nel caso di una dll contenente il prodotto della precompilazione delle view, l'uso di `AssemblyPart` sarà superfluo, anche se trascurabile.

Fatta questa modifica, otterremo un applicativo il cui comportamento è dinamico a seconda delle librerie inserite nella directory plugin che vengono caricate in autonomia, senza alcuna distinzione rispetto a quelle referenziate direttamente nel progetto.

I middleware e il contesto HTTP

Nel Capitolo 3 abbiamo introdotto il concetto dei middleware e abbiamo parlato di come siano fondamentali nel processo di gestione di una richiesta. Sono un elemento fondamentale di ASP.NET, perché è solo grazie ai middleware che riusciamo a servire file statici, supportare le funzionalità di routing e di MVC e proteggere le nostre pagine con autenticazione e autorizzazione.

La modularità di ASP.NET è così elevata che un applicativo può anche essere privato di ogni meccanismo e darci comunque il controllo totale di una richiesta HTTP. Nell'Esempio 14.11 possiamo vedere il più semplice degli applicativi che possiamo sviluppare, che otteniamo se in Visual Studio creiamo un progetto ASP.NET Core Web Application di tipo vuoto.

Esempio 14.11

```
public void Configure(IApplicationBuilder app)
{
   app.Run(async (context) =>
   {
      await context.Response.WriteAsync("Hello World!");
   });
}
```

Nell'esempio possiamo notare l'uso della funzione `Run` per configurare un delegato che riceve un'istanza di `HttpContext` (nell'esempio di nome `context`) e tramite quest'ultima scrive in risposta. Ogni operazione che facciamo sulla risposta avviene in asincrono, perciò sfruttiamo il pattern `async/await` per trarne il massimo dei benefici. L'oggetto `HttpContext`, esposto anche quando siamo in un controller MVC, è il fulcro di ogni richiesta HTTP e contiene i dettagli sulla richiesta, sull'utente, sulla connessione e ci dà accesso alla risposta. Nella Tabella 14.1 sono elencati i principali membri presenti, per fornire un'idea di cosa possiamo fare.

Tabella 14.1 – Principali membri dell'oggetto HttpContext.

Proprietà	Descrizione
`Connection`	Restituisce un oggetto che dà informazioni sulla connessione fisica, sugli IP e i certificati coinvolti
`Request`	Restituisce un oggetto di tipo HttpRequest contenente tutte le informazioni della richiesta
`RequestAborted`	Restituisce un CancellationToken che permette di supportare la cancellazione delle richieste asincrone
`RequestServices`	Restituisce il riferimento a IServiceProvider per poter risolvere le dipendenze esplicitamente
`Response`	Restituisce un oggetto di tipo HttpResponse che permette di rispondere all'utente
`Session`	Restituisce un oggetto che permette di memorizzare informazioni di stato
`User`	Restituisce informazioni sull'utente corrente in base al sistema di autenticazione

L'oggetto `HttpRequest` è l'oggetto da guardare se vogliamo accedere ai cookie, alla querystring e alla form, senza l'ausilio di model binding forniti da ASP.NET MVC. Nella Tabella 14.2, proponiamo i membri più importanti, il cui utilizzo è piuttosto intuitivo.

Tabella 14.2 – Principali membri dell'oggetto HttpRequest.

Proprietà	Descrizione
Body	Restituisce lo stream di byte della richiesta HTTP ricevuta e da gestire

Cookies	Restituisce un dizionario di coppie chiave/valore contenente i cookie r icevuti
Form	Restituisce un dizionario che interpreta il Body come una form, permettendoci di accedere tramite coppie chiave/valore
Headers	Restituisce un dizionario di coppie chiave/valore contenente gli header ricevuti
Method	Restituisce il metodo HTTP della richiesta
Path	Restituisce il path relativo della richiesta, per esempio /home
Query	Restituisce un dizionario di coppie chiave/valore contenente i parametri in querystring presenti nell'indirizzo della richiesta

Per quanto riguarda la risposta, i membri essenziali per poter rispondere all'utente sono indicati nella Tabella 14.3.

Tabella 14.3 – Principali membri dell'oggetto HttpResponse.

Membri	Descrizione
Body	Restituisce lo stream di byte nel quale possiamo scrivere la risposta
ContentType	Permette di scrivere l'header Content-Type della risposta
Cookies	Restituisce un oggetto che ci permette di aggiungere o cancellare cookie sulla risposta
Headers	Restituisce un dizionario di coppie chiave/valore che ci permette di scrivere header in uscita
StatusCode	Permette di scrivere lo status code numerico della risposta
Redirect	Imposta lo status code a 302 e l'header di Location per rimandare l'utente a un altro indirizzo
RegisterForDispose	Permette di passare un delegato da invocare quando la risposta è completata

Ai membri esposti da questi principali oggetti troviamo inoltre alcuni extension method che facilitano l'accesso ad alcuni di essi o ne arricchiscono le funzionalità. Per esempio, tramite il namespace `Microsoft.AspNetCore.Http.Extensions` troviamo la funzione `GetEncodedUrl` che restituisce l'URI completo della risposta. Oppure, tramite il namespace `Microsoft.AspNetCore.Http` troviamo la funzione `WriteAsync` utilizzata nell'Esempio 14.11. In questi casi consigliamo di fare affidamento all'IntelliSense di Visual Studio, per poter sfruttare appieno tutte le funzionalità.

Visti gli oggetti fondamentali con i quali dobbiamo lavorare, possiamo ritornare al middleware dell'Esempio 14.1. Grazie alla funzione `Run` indichiamo un middleware completamente personalizzato, ma è l'unico presente nell'applicativo.

Creare un middleware personalizzato

L'oggetto `IApplicationBuilder` sul quale andiamo a configurare i middleware dispone di alcune funzioni per configurare delegati nei quali intervenire nella richiesta. La funzione `Use` ci permette di indicare una funzione asincrona da invocare all'interno della pipeline, espone `HttpContext` ed è l'ideale per scrivere piccoli snippet di codice.

Esempio 14.12

```
app.Use(async (context, next) =>
{
   await context.Response.WriteAsync("Middleware 1 pre");
   // Chiamo il middleware successivo
   await next();
   await context.Response.WriteAsync("Middleware 1 post");
});

app.UseDeveloperExceptionPage();

app.Use(async (context, next) =>
{
   await context.Response.WriteAsync("Middleware 2");
   // Chiamo il middleware successivo
   await next();
});
```

Rispetto a quanto abbiamo visto nell'Esempio 14.11, la funzione `Use` può essere chiamata più volte e nella pipeline di esecuzione otteniamo l'invocazione dei middleware nell'ordine in cui li abbiamo definiti. Il secondo parametro `next` che le funzioni ricevono dà la possibilità di invocare il middleware successivo, perciò ci dà la facoltà di decidere se lavorare prima o dopo i middleware successivi (che a loro volta possono chiamare i successivi), semplicemente invertendo le istruzioni. Non solo: tramite i costrutti `try/catch/finally`

possiamo intervenire inibendo eventuali errori sull'invocazione del next o eseguendo operazioni indipendentemente dall'esito degli altri middleware.

Nell'Esempio 14.11 abbiamo volutamente inserito la chiamata a `UseDeveloperExceptionPage` in mezzo ai nostri due middleware: nel caso di errori la visualizzazione di una pagina, avviene solo se questi sono generati dal secondo e dai successivi middleware. Alla luce di tutte queste considerazioni, richiamando qualsiasi indirizzo del web server, otterremo a video il seguente testo.

Esempio 14.13

```
Middleware 1 pre
Middleware 2
Middleware 1 post
```

È chiaro che rispondere sempre con lo stesso contenuto non trova utilità, soprattutto senza considerare le informazioni della richiesta. L'accesso a `HttpContext` ci permette di vedere la richiesta e quindi la relativa proprietà `Path`, ma le possibilità sono molteplici, e vanno dalla completa scrittura della risposta fino a piccoli comportamenti, come l'aggiunta di un header personalizzato.

Oltre alla funzione `Use` disponiamo di un'altra funzione simile, di nome `Map`, la quale ci permette di distinguere su quale percorso creare una branch che determina una nuova pipeline di middleware di esecuzione.

Nell'Esempio 14.14 possiamo vedere come sfruttare tale istruzioni per richiedere che all'interno di un percorso `/secret` si possa accedere solo in HTTPS, restituendo un 403 (Forbidden) in sua assenza.

Esempio 14.14

```
// Use*** middleware precedenti

app.Map("/secret", b =>
{
   // Middleware del branch
   b.Use((context, next) =>
   {
      // Verifica HTTPS
      if (!context.Request.IsHttps)
      {
         context.Response.StatusCode = 403;
         return Task.CompletedTask;
      }
      // Chiamo il middleware successivo
      return next();
   });
   b.UseStaticFiles();
});

// Use*** middleware successivi
```

Il percorso richiesto dalla funzione `Map` dev'essere relativo e determina una nuova pipeline. Questo significa che i middleware configurati prima di essa vengono normalmente chiamati, mentre non rientrano quelli successivi. È per questo motivo che all'interno del delegato riceviamo un'istanza di `IApplicationBuilder` (parametro `b`) con il quale costruire un pezzo della pipeline totale. La chiamata a `UseStaticFiles` è quindi fondamentale, anche se questa viene già fatta, ma sul builder principale, successivamente a `Map`. Non commettiamo quindi l'errore di anteporre i middleware che vogliamo siano condivisi, perché dobbiamo sempre considerare il corretto ordine di esecuzione. `UseStaticFiles`, per esempio, non può essere posto prima di `Map`, perché verrebbe eseguito prima della verifica del protocollo. `Map` è quindi da usare solo nelle situazioni in cui vogliamo differenziare notevolmente la linea dei middleware da eseguire. Un aspetto interessante, infine, deriva dal fatto che le chiamate a `Map` possono essere innestate l'una all'altra, poiché stiamo sempre lavorando con un'istanza di `IApplicationBuilder`.

Molto simile a `Map` è `MapWhen`, che invece di un percorso vuole un predicato che ci permette in modo libero di valutare i parametri della richiesta. Nell'Esempio 14.15 sfruttiamo questa possibilità per fornire una pagina di stato solo se chi effettua la richiesta è un browser locale e solo per un percorso specifico.

Esempio 14.15

```
app.MapWhen(c =>
   // Solo IP locale
   System.Net.IPAddress.IsLoopback(c.Connection.RemoteIpAddress)
   // Solo /health
   && c.Request.Path.StartsWithSegments("/health"),
   b =>
{
   // Unico middleware eseguito
   b.Run(context => context.Response.WriteAsync("Status: OK"));
```

```
});
```

L'uso delle lambda è facoltativo e pertanto potremmo fare uso di funzioni separate, anche se in questo caso risulta più appropriato sviluppare un middleware dedicato.

Creare un middleware come componente

Quando le logiche dei nostri middleware diventano complesse e, soprattutto, quando vogliamo poterle riutilizzare, ha senso sviluppare un componente che ci permetta di adottarle facilmente, allo stesso modo di come sfruttiamo quelli di ASP.NET.

Il primo passo da compiere consiste nel creare una classe che contenga due requisiti: la presenza di un costruttore che accetta un delegato e la presenza di una funzione di nome `Invoke`. Nell'Esempio 14.16 possiamo vedere la struttura base.

Esempio 14.16

```
public class TimerMiddleware
{
   private readonly RequestDelegate _next;
   public TimerMiddleware(RequestDelegate next)
   {
      _next = next;
   }
   public async Task Invoke(HttpContext httpContext)
   {
      await _next();
   }
}
```

Notiamo che gli stessi elementi richiesti dalla funzione `IApplicationBuilder.Use` sono presenti anche qua. Il delegato che ci permette di invocare il middleware successivo viene passato nel costruttore, mentre il contesto sulla funzione `Invoke`. Questa differenziazione nasce dal fatto che per una questione di ottimizzazione la classe viene istanziata una sola volta, mentre il metodo `Invoke` viene chiamato a ogni richiesta, con un contesto diverso. Ciò significa che dobbiamo rendere atomica la funzione e non depositare informazioni di stato sui cambi della classe.

L'assenza di un'interfaccia è dovuta, per permetterci di essere flessibili nella firma del costruttore e della funzione, perché su entrambi godiamo della possibilità di ricevere oggetti tramite la dependency injection. È sufficiente aggiungere le proprie dipendenze a seconda del ciclo di vita che hanno, tipicamente singleton sul costruttore e scope sulla funzione, come è mostrato nell'Esempio 14.17.

Esempio 14.17

```
private IMyService _service;

public TimerMiddleware(RequestDelegate next, IMyService service)
{
   _service = service;
   _next = next;
}

public async Task Invoke(HttpContext httpContext, MyDbContext context)
{
   // ...
```

In alternativa, l'oggetto `HttpContext` ci dà accesso tramite la proprietà `RequestServices` all'`IServiceProvider` permettendoci di ottenere gli oggetti che vogliamo dal motore di dependency injection.

Supponendo di realizzare un middleware che misura i tempi di esecuzione di ogni richiesta, scrivendoli nell'header di uscita, un'ipotetica implementazione potrebbe essere quella mostrata nell'Esempio 14.18.

Esempio 14.18

```
public async Task Invoke(HttpContext httpContext)
{
   // Inizio a cronometrare
   Stopwatch stopwatch = Stopwatch.StartNew();
   try
   {
      await _next(httpContext);
   }
   finally
   {
      // Fermo il cronometro
      stopwatch.Stop();
```

```
      // Controllo che la risposta non sia già stata mandata
      if (!httpContext.Response.HasStarted)
      {
         // Scrivo l'header
         httpContext.Response.Headers["x-execution-time"] =
            stopwatch.Elapsed.ToString("g");
      }
   }
}
```

L'uso del costrutto `try/finally` ci permette di inserire sempre il nostro header, indipendentemente dall'esito dell'operazione. Risulta fondamentale l'uso della proprietà `HasStarted`, per controllare che la risposta non sia già stata inviata al client, anche parzialmente. Questa situazione si può verificare in presenza di middleware che scrivono file o che forzano la scrittura della risposta senza buffer. Quando questo avviene, scrivere un header genererebbe un errore, situazione che vogliamo quindi evitare.

Una volta realizzato il middleware, non ci resta che utilizzarlo invocando `IApplicationBuilder.UseMiddleware`, il cui scopo è registrare il tipo, ma seguendo lo stile di ASP.NET possiamo realizzare un extension method che renda più comodo l'utilizzo, come quello mostrato nell'Esempio 14.19.

Esempio 14.19

```
namespace Microsoft.AspNetCore.Builder
{
   public static class TimerExtensions
   {
      public static IApplicationBuilder UseTimer(
         this IApplicationBuilder applicationBuilder)
      {
         return applicationBuilder.UseMiddleware<TimerMiddleware>();
      }
   }
}
```

Con questa definizione possiamo chiamare `app.UseTimer` allo stesso modo di come chiamiamo `app.UseMvc`, facendo solo attenzione all'ordine di registrazione. L'uso del namespace relativo al builder facilita ulteriormente la visibilità dell'extension method e il suo utilizzo. Quello che abbiamo scritto è un middleware semplice, ma vediamo di capire come sfruttarlo in modo più avanzato.

Concetti avanzati sui middleware

I middleware sono un componente fondamentale, non solo dal punto di vista del ruolo che ricoprono ma anche dal punto di vista delle prestazioni. È importante scrivere il componente affinché non ci siano memory leak e non vengano creati oggetti inutili. L'oggetto `HttpContext` dispone della funzione `RegisterForDispose` utile al fine del `Dispose` corretto degli oggetti. Possiamo utilizzarlo come alternativa al costrutto `try/finally`, soprattutto se vogliamo essere certi di effettuare il `Dispose` di un oggetto solo a risposta scritta e non circoscritto alla propria esecuzione, come viene mostrato nell'Esempio 14.20.

Esempio 14.20

```
public async Task Invoke(HttpContext httpContext)
{
   Stream stream = await OpenStreamAsync(httpContext);
   httpContext.Response.RegisterForDispose(stream);
  // ...
```

Una proprietà fondamentale da sfruttare è `RequestAborted`, sempre dell'oggetto `HttpContext`, poiché essa ci dà accesso al `CancellationToken` della richiesta dell'utente. Se la connessione viene interrotta, viene attivato il segnale e, se questo è correttamente gestito, possiamo di conseguenza annullare l'operazione asincrona in corso. È buona norma, infatti, implementare il pattern di cancellazione all'interno di ogni funzione asincrona, ricevendo un `CancellationToken`. Tutti i membri asincroni di .NET accettano questo oggetto per interrompere, se necessario, le comunicazioni socket o operazioni sull'I/O. L'Esempio 14.21 mostra come sfruttare il token per annullare una chiamata HTTP sfruttando la proprietà indicata.

Esempio 14.21

```
public async Task Invoke(HttpContext httpContext,
   IHttpClientFactory clientFactory)
{
   HttpClient client = clientFactory.CreateClient();
   // Passo il token
   await client.GetAsync("http://www.google.com",
      httpContext.RequestAborted);
```

```
    // Raise dell'eccezione in caso di cancellazione
    httpContext.RequestAborted.ThrowIfCancellationRequested();
   // ...
```

Una volta ottimizzato il codice del middleware, il passo successivo può essere quello di integrarci con la pipeline per permettere un dialogo con gli altri middleware. L'oggetto `HttpContext` dispone a tal fine una proprietà `Features`, una collezione di oggetti popolati in parte dal web server e in parte dai middleware, e rappresenta il punto centrale, contenitore di tutte le informazioni della richiesta che viene soddisfatta, tant'è che oggetti come `HttpRequest` e `HttpResponse` non sono altro che wrapper sulle feature, per facilitarne l'accesso.

Per il nostro esempio definiamo quindi una feature, in genere rappresentata tramite un'interfaccia, dando accesso allo `Stopwatch` della richiesta corrente.

Esempio 14.22

```
public interface ITimerFeature
{
   Stopwatch Stopwatch { get; }
}

public class TimerFeature : ITimerFeature
{
   public Stopwatch Stopwatch { get; }
   public TimerFeature(Stopwatch stopwatch)
   {
      Stopwatch = stopwatch;
   }
}
```

La relativa implementazione non fa altro che esporre il cronometro come da noi richiesto. Nel middleware non ci resta che inserire la nostra feature all'interno del contesto, come viene mostrato nell'Esempio 14.23.

Esempio 14.23

```
public async Task Invoke(HttpContext httpContext)
{
   Stopwatch stopwatch = Stopwatch.StartNew();
   // Aggiungo la feature del timer
   ITimerFeature feature = new TimerFeature(stopwatch);
   httpContext.Features.Set(feature);
}
```

La feature può essere letta da chiunque abbia accesso al contesto e verrà eliminata con esso. Anche in questo caso, un extension method può venire in aiuto per facilitare l'accesso.

Esempio 14.24

```
public static class TimerExtensions
{
   public static Stopwatch GetTimer(this HttpContext httpContext)
   {
      return httpContext.Features.Get<ITimerFeature>()?.Stopwatch;
   }
}
```

Con la presenza di questa funzione, chiunque abbia accesso al contesto, come una action di ASP.NET MVC, può accedere allo `Stopwatch` per effettuare operazioni su di esso.

Viste queste caratteristiche dedicate ad ASP.NET in generale, passiamo a qualcosa di più specifico della parte MVC, per capire come inserire aspetti di Aspect Oriented Programming, tramite i filtri.

Applicare filtri ai controller MVC

MVC è una parte dell'intero framework ASP.NET, che viene azionata attraverso un middleware che porta la richiesta a un livello più avanzato, dove viene mappata su un controller, è identificata una action e i parametri sono convertiti in modelli; il tutto per restituire infine un risultato che molto spesso è una view che genera HTML.

Quando la palla passa a MVC, si aziona una pipeline di invocazione, che dal punto di vista logico è molto vicina ai middleware, ma è più specifica nei confronti dei concetti MVC. Essa è caratterizzata da filtri che vengono invocati prima e dopo l'esecuzione di una azione e possono essere di natura diversa a seconda delle competenze e del momento in cui vengono invocati e sono:

- Authorization filter: i primi tra tutti a essere invocati, hanno il compito di determinare se l'utente corrente è autorizzato.

- Resource filter: vengono invocati a seguito dell'autorizzazione, prima dell'azione del controller e dopo di essa, a seguito di tutti gli altri filtri. Vengono avviati prima del model binding e sono l'ideale per implementare meccanismi di cache che inibiscono l'invocazione della action originale.

- Action filter: vengono invocati subito prima e dopo la action di un controller e permettono di manipolare gli argomenti della action o il risultato restituito, eventualmente inibendo la chiamata alla action originale.

- Exception filter: vengono eseguiti solo quando si verifica un'eccezione non gestita da parte della action o dall'esecuzione del risultato della stessa.

- Result filter: sono eseguiti solo quando la action ha avuto successo e ha restituito un risultato. Con i Result filter, possiamo intercettare l'esecuzione prima e dopo il risultato.

L'ordine di esecuzione dei filtri e il momento in cui lavorano è visibile nella Figura 14.2.

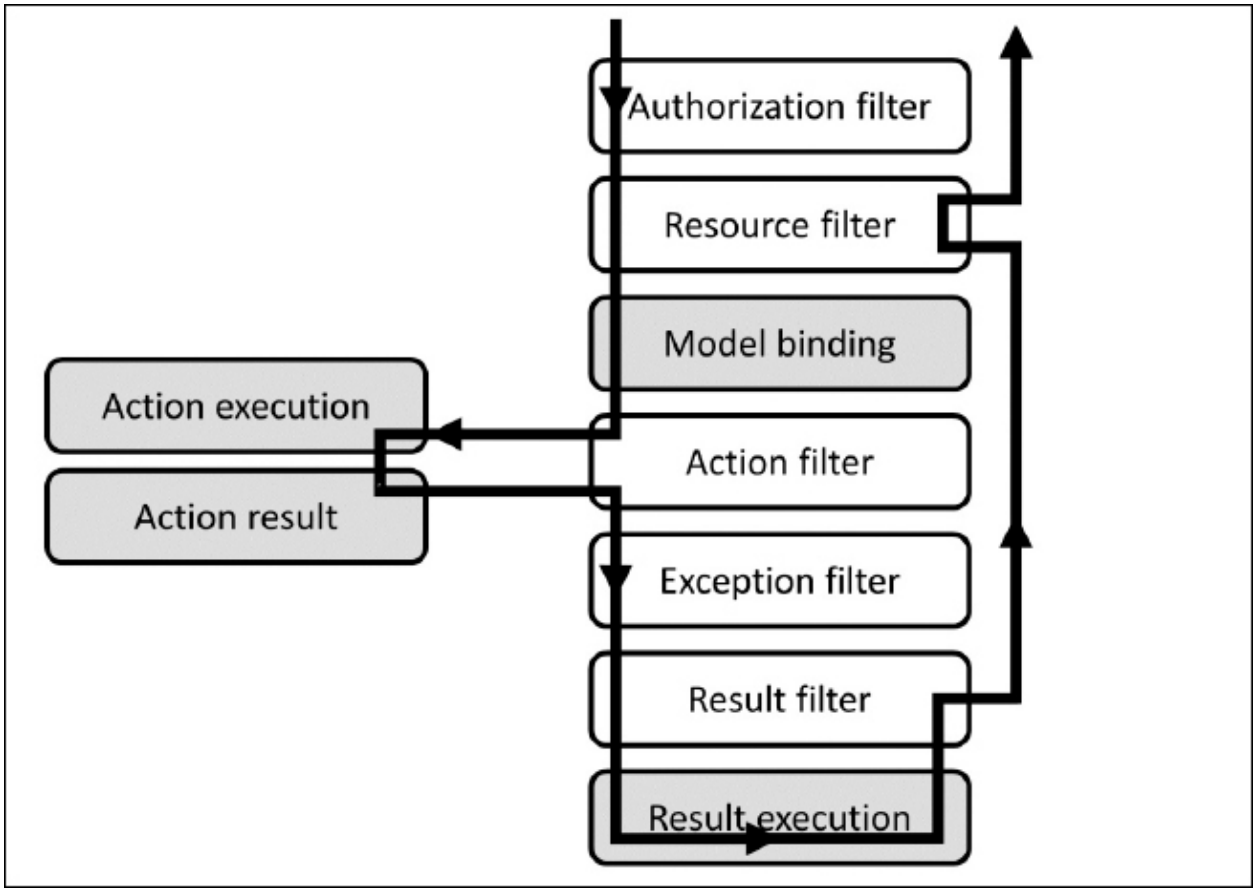


Figura 14.2 – Tipologie di filtri e ordine di esecuzione nella pipeline MVC.

I filtri sono ideali per poter inserire velocemente comportamenti del nostro applicativo e sono la base di molte funzionalità che affronteremo in questo capitolo. Sono oggetti di tipo `IFilterMetadata` nella loro rappresentazione base e dispongono di specializzazioni a seconda della loro tipologia, i cui nomi sono auto-esplicativi: `IAuthorizationFilter`, `IResourceFilter`, `IActionFilter`, `IExceptionFilter` e `IResultFilter`. Per ognuna di esse, esiste una controparte `IAsync***Filter` che espone le medesime funzionalità ma in modo asincrono, restituendo un `Task`.

Per utilizzarli dobbiamo popolare la collezione `Filters` delle opzioni disponibili in fase di configurazione di MVC, come viene mostrato nell'Esempio 14.25.

Esempio 14.25

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc(o =>
```

```
        {
           o.Filters.Add(new RequireHttpsAttribute());
        });
 }
```

Nell'esempio utilizziamo un filtro di tipo autorizzativo, che si accerta che ogni richiesta fatta a MVC avvenga tramite HTTPS e, in caso negativo, effettuerà un redirect. In alternativa al popolamento della collezione `Filters`, possiamo sfruttare gli attributi di .NET, marcando in modo più mirato i controller o le action, a seconda del grado di controllo che vogliamo avere. Il filtro usato nell'Esempio 14.25, oltre a implementare `IAuthorizationFilter` è anche un `Attribute` che possiamo applicare in modo specifico, come viene mostrato nell'Esempio 14.26.

Esempio 14.26

```
[RequireHttps]
public class AccountController : Controller
{
   [ResponseCache(CacheProfileName = "30sec")]
   public IActionResult Index()
   {
      return View();
   }
```

Abbiamo già visto in atto questa tecnica quando abbiamo sfruttato l'attributo `ResponseCacheAttribute` nel Capitolo 8, per effettuare la cache delle response mediante appunto un `IActionFilter` già messo a disposizione.

Nella Figura 14.2 abbiamo visto che ogni tipologia di filtro identifica una fase specifica dove intervenire e determina l'ordine di esecuzione, importante soprattutto se siamo in presenza di più filtri definiti a livello globale, di controller e di action. In caso di più filtri della stessa tipologia, essi vengono eseguiti nell'ordine in cui sono stati inseriti, che nel caso di attributi è a livello globale, di controller e di action. Poiché alcuni di questi filtri sono in grado di lavorare prima e dopo l'esecuzione della action, essi eseguono la post action in ordine inverso una volta che la result è stata eseguita, con la stessa logica che contraddistingue le tipologie filtri, come si evince dalla Figura 14.2. Qualora questa logica non ci dovesse soddisfare, ogni attributo in genere implementa anche l'interfaccia `IFilterOrder` che, tramite la proprietà `Order`, ci permette di dare una priorità ai figli attraverso un numero con il quale viene poi fatto un ordinamento ascendente, eseguendo per primi i numeri più bassi.

Ora che sappiamo cosa sono e come utilizzare i filtri, non ci resta che vedere come procedere con una implementazione personalizzata.

Creare un filtro personalizzato

Per implementare un filtro personalizzato, teoricamente dovremmo implementare una delle interfacce di specializzazione di `IFilterMetadata`, ma ci vengono in aiuto alcune classi già pronte all'uso. La più comune da usare è la `ActionFilterAttribute`, la quale implementa già per noi un attributo, utile per marcare controller e action, e le interfacce `IActionFilter`, `IAsyncActionFilter`, `IResultFilter`, `IAsyncResultFilter`, `IOrderedFilter`. Contiene tutto quello che serve per intercettare un'azione e il risultato, il tutto in modo sincrono o asincrono. Non dobbiamo far altro che scegliere il momento da intercettare ed eseguire l'override di uno o più membri tra quelli disponibili nella Tabella 14.5.

Tabella 14.4 – Metodi da sovrascrivere per implementare un filtro.

Metodi	Descrizione
`OnActionExecuting`	Permette di intercettare la action di un controller prima che venga eseguita
`OnActionExecuted`	Permette di gestire il risultato di una action del controller
`OnResultExecuting`	Permette di intercettare l'esecuzione del risultato ottenuto dalla action
`OnResultExecuted`	Permette di gestire la risposta prodotta dall'esecuzione di un risultato
`OnActionExecutionAsync`	Permette di intercettare in modo asincrono l'esecuzione di una action, intervenendo prima e dopo
`OnResultExecutionAsync`	Permette di intercettare in modo asincrono l'esecuzione di un risultato, intervenendo prima e dopo

Poniamo quindi il caso di voler realizzare un filtro che inibisce l'accesso a una pagina quando l'utente è anonimo e nelle ore notturne. Per farlo, dobbiamo intervenire prima che l'azione originale venga chiamata e interrompere il proseguimento della pipeline, se è necessario. L'Esempio 14.27 mostra come sovrascrivere `OnActionExecuting`, per poter personalizzare la proprietà `Result` del contesto, al fine di annullare l'esecuzione della action e del suo risultato, fornendone uno personalizzato.

Esempio 14.27

```
public class TimeAttribute : ActionFilterAttribute
{
   public int Hour { get; set; } = 6;
   public override void OnActionExecuting(ActionExecutingContext context)
   {
      bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
      bool outOfTime = DateTime.Now.Hour < Hour;
      if (anonymous && outOfTime)
      {
         context.Result = new ViewResult { ViewName = "OutOfTime" };
      }
   }
}
```

Osserviamo come il metodo sovrascritto riceva un contesto che dà accesso all'`HttpContext` visto in precedenza nel capitolo. Possiamo vedere nell'Esempio 14.28 un'implementazione complementare asincrona, che nel nostro caso non porta alcun beneficio poiché sono assenti chiamate asincrone.

Esempio 14.28

```
public override Task OnActionExecutionAsync(
   ActionExecutingContext context,
   ActionExecutionDelegate next)
{
   bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
   bool outOfTime = DateTime.Now.Hour < Hour;
   if (anonymous && outOfTime)
   {
      context.Result = new ViewResult { ViewName = "OutOfTime" };
      return Task.CompletedTask;
   }
   // Chiamo il prossimo filtro o l'esecuzione della action
   return next();
}
```

Non vi sono differenze degne di nota a seconda del tipo di filtro che possiamo implementare e, nella maggior parte dei casi, abbiamo accesso al contesto e alla manipolazione del risultato. Ciò che può risultare utile, invece, è l'integrazione dei filtri con la dependency injection.

Utilizzare la dependency injection con i filtri

L'uso degli attributi per indicare dove applicare il filtro è molto comodo e intuitivo, ma presenta un difetto: è presente un'unica istanza dell'attributo per ogni utilizzo. Questo significa che nella classe non possiamo fare affidamento a membri privati per memorizzare informazioni, magari tra l'executing e l'execution. Oltre a questo non possiamo sfruttare la dependency injection per ottenere servizi utili ai nostri scopi, se non passando dall'oggetto `HttpContext`, il quale espone un'istanza di `IServiceProvider`.

Per ovviare a questo fatto, abbiamo a disposizione due attributi particolari, dei filtri che fanno da tramite e sfruttano la dependency injection. Il primo attributo, di nome `ServiceFilterAttribute`, ci permette di specificare il tipo da istanziare tramite dependency injection.

Esempio 14.29

```
[ServiceFilter(typeof(TimeAttribute))]
public IActionResult Index()
{
   return View();
}
```

Per far sì che tutto funzioni, è necessario che il container conosca il tipo `TimeAttribute`, perciò dobbiamo registrarlo nello `Startup.cs` all'interno del metodo `ConfigureServices`. Grazie alla dependency injection, possiamo arricchire il costruttore del nostro filtro, richiedendo servizi terzi. In questo modo, a ogni richiesta che coinvolge la action dell'Esempio 14.29 viene istanziata la classe `TimeAttribute` con le necessarie dipendenze, per poi essere cestinata. Il difetto di questa soluzione consiste nel fatto che non possiamo specificare parametri personalizzati, come per esempio dei valori primitivi non presenti nel container.

In alternativa possiamo usare l'attributo `TypeFilterAttribute`, che si differenzia dal fatto che non richiede che il filtro sia registrato e consente l'uso di argomenti personalizzati misti ad argomenti da risolvere tramite la dependency injection. Nell'Esempio 14.30, possiamo vedere un esempio in cui indichiamo di utilizzare il filtro passando il numero delle ore al costruttore.

Esempio 14.30

```
[TypeFilter(typeof(TimeAttribute), Arguments = new object[] { 7 })]
public IActionResult Index()
{
   return View();
}
```

Questo secondo attributo è senz'altro più potente, ma complica un po' l'utilizzo dell'attributo, perciò possiamo valutare un ultimo approccio, che richiede del codice in più, ma più completo e di facile utilizzo.

Possiamo sfruttare un'altra interfaccia di nome `IFilterFactory`, che eredita da `IFilterMetadata`, per implementare un generatore di filtri che può essere sfruttato a livello globale o come attributo. Creiamo quindi un attributo che implementa solo questa interfaccia, come nell'Esempio 14.31.

Esempio 14.31

```
public class Time2Attribute : Attribute, IFilterFactory
{
   public int Hour { get; set; }
   public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)
   {
   //IMyService service = serviceProvider.GetRequiredService<IMyService>();
   return new TimeFilter(Hour);
   }
   public bool IsReusable => false;
}
```

Come per qualsiasi attributo, esponiamo una proprietà di configurazione, in questo caso di nome `Hour`, che possiamo poi sfruttare nella creazione del filtro tramite `CreateInstance`. Se finora i filtri sono stati al tempo stesso un attributo e svolgevano la duplice funzione di intervenire nella pipeline e di servire alla loro applicazione, con questa tecnica separiamo i ruoli. All'atto della creazione del filtro siamo noi a decidere come creare l'oggetto e, al tempo stesso, disponiamo dell'`IServiceProvider` corrente, che ci permette di risolvere anche dipendenze terze. Non ci resta che implementare `TimeFilter`, limitandoci alla sola implementazione di `IActionFilter` poiché solo a esso siamo interessati.

Esempio 14.32

```
public class TimeFilter : IActionFilter
{
   public int Hour { get; }
   public TimeFilter(int hour)
   {
      Hour = hour;
   }
   public void OnActionExecuting(ActionExecutingContext context)
   {
      bool anonymous = !context.HttpContext.User.Identity.IsAuthenticated;
      bool outOfTime = DateTime.Now.Hour < Hour;
      if (anonymous && outOfTime)
      {
         context.Result = new ViewResult { ViewName = "OutOfTime" };
      }
   }
   public void OnActionExecuted(ActionExecutedContext context)
   {
      // Niente da fare
   }
}
```

Come possiamo vedere, abbiamo solo spostato l'implementazione del filtro dall'attributo verso una classe separata. Questa classe, di conseguenza, può essere utilizzata tramite l'attributo o direttamente istanziata in fase di configurazione dell'applicativo per andare a popolare la collezione `Filters`.

Conclusioni

ASP.NET Core gode di una modularità che ci permette di personalizzare ogni suo aspetto. In questo capitolo abbiamo visto come sfruttare questa caratteristica per sviluppare librerie che s'inseriscono automaticamente nello startup dell'applicazione o forniscono contenuti come view Razor. Questa caratteristica facilita l'organizzazione dello sviluppo in team e la capacità di riutilizzare i componenti su più

progetti. Con la stessa prospettiva, abbiamo visto come realizzare middleware personalizzati che sappiano leggere la richiesta e processare la risposta, in armonia con gli altri configurati. Le feature ci permettono di entrare nelle dinamiche di processamento della richiesta e condividere informazioni con tutti i componenti coinvolti, fino ad arrivare alla view.

Infine, siamo entrati nello specifico della parte MVC e abbiamo analizzato i filtri, e come possiamo configurarli a livello globale o nello specifico di un controller o di un action. Con essi possiamo aggiungere comportamenti in modo flessibile e riutilizzare logiche semplicemente applicando un attributo. Nel prossimo capitolo continueremo questo viaggio di approfondimento sull'estendibilità di ASP.NET Core, esaminando altre caratteristiche del framework MVC.

15
Model binder e tag helper personalizzati

Nel capitolo precedente siamo entrati nel dettaglio di ASP.NET Core per capire meglio come funziona e come poter intraprendere un percorso che ci porti a estendere e creare componenti utili ai nostri applicativi.

Vogliamo quindi proseguire questo viaggio restando sempre nel tema di ASP.NET Core MVC, che rappresenta una delle caratteristiche più importanti del framework. Partiremo entrando nel dettaglio del model binding, cioè della capacità di ASP.NET MVC di astrarre dalle richieste HTTP e ragionare con i modelli. Vedremo poi come attorno a essi ruotano meta informazioni che vengono associate e poi lette dalle view, dal binding e dai validatori per poter offrire tutte le funzionalità che rendono efficace questo pattern architetturale. Finiremo poi con l'approfondire il tema dei tag helper, per capire come sviluppare porzioni di logiche HTML riutilizzabili, che sfruttino anche queste meta informazioni.

Il model binding

Il livello di astrazione introdotto da ASP.NET Core MVC porta notevoli benefici, tra cui il fatto che è venuta meno la necessità di lavorare direttamente con le richieste HTTP: tutte le richieste in ingresso vengono rimandate dal meccanismo di routing al metodo del controller corrispondente, ma gli eventuali parametri a essa associati vengono ricostruiti da un altro componente del framework, chiamato model binder, il cui compito è proprio quello di mappare parametri presenti in diverse sorgenti dati, come querystring o body, in oggetti. Lo abbiamo già visto in azione nei capitoli precedenti e lo abbiamo spesso dato per scontato perché ASP.NET Core di default è in grado di fare un ottimo lavoro per ricostruire gli oggetti, ma all'interno di questo capitolo ne discuteremo nel dettaglio per capirne meglio il funzionamento.

Il routing viene definito, solitamente, all'interno della classe `Startup` nel metodo `Configure` e, se creato tramite il template standard, contiene un mapping simile a quello illustrato nell'Esempio 15.1.

Esempio 15.1

```
app.UseMvc(routes =>
{
   routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
});
```

Senza entrare troppo nel dettaglio, considerando anche che il routing è già stato discusso minuziosamente all'interno del Capitolo 5, nell'Esempio 15.1 viene definita una route il cui template è composto da tre segmenti: il controller, la action e un parametro, chiamato `id`, definito come opzionale tramite il carattere '`?`'. Questo – a patto che ci sia una action all'interno del controller che lo permetta – garantisce la corretta esecuzione di tutte le chiamate elencate nell'Esempio 15.2.

Esempio 15.2

```
https://www.miosito.com/
https://www.miosito.com/Home/
https://www.miosito.com/Home/Index/
https://www.miosito.com/Home/Index/1
https://www.miosito.com/Home/Index/abc
```

La action che è in grado di rispondere a queste chiamate è evidenziata nell'Esempio 15.3.

Esempio 15.3

```
public class HomeController : Controller
{
   public IActionResult Index(string id)
   {
      return View();
   }
}
```

Il parametro `id`, il cui tipo in questo caso è `string`, verrà popolato in automatico, grazie al model binder, con il valore recuperato dal terzo segmento della route `default` creata in precedenza. Quando viene passato un tipo non corrispondente, come il valore `1` mostrato nell'Esempio 15.2, il model binder si interroga e lavora con il routing per capire se esiste una action che può accettare il tipo intero passato. Si accorgerà dell'esistenza di una sola action il cui parametro richiesto è, nonostante il nome identico, di un tipo diverso, e quindi cercherà di capire se è in grado di fare la conversione di tipo: poiché per passare da intero a stringa è sufficiente una chiamata al metodo `ToString`, il model binder inietterà nella action `Index` il valore e il tipo corretto. Qualora invece non sia presente il valore, come per la route `/Home/Index`, verrà utilizzato il valore di default previsto dal tipo del parametro della action: zero per valori interi, `null` per reference type e così via.

Come abbiamo già anticipato nei capitoli precedenti e dimostrato in questi esempi introduttivi, ci sono due dettagli a cui bisogna prestare particolare attenzione: il primo è il nome del parametro, che deve corrispondere a quanto specificato nella route, altrimenti non verrà ricostruito dal model binder, mentre il secondo è l'ordine di elaborazione delle varie sorgenti tramite il quale il parametro viene prelevato. Nonostante a livello di controller e action risulti tutto trasparente, il parametro `id` mostrato nell'Esempio 15.3 viene ricercato e ricostruito a partire da sorgenti diverse, quali:

- Form: i valori vengono recuperati dalle chiamate marcate HTTP POST.
- Route: i valori sono elencati nel template di routing.
- Querystring: i valori sono opzionali, fanno parte dell'URL e sono identificati dal carattere '?'.

L'ordine di caricamento è proprio quello appena elencato, ed è importante conoscerlo: se la chiamata a `/Home/Index` fosse stata in `POST` e ci fosse stato un parametro chiamato `id` nella form inviata, questo avrebbe preso la precedenza rispetto a quanto dichiarato a livello di route e quindi il valore effettivo sarebbe andato perso.

In riferimento al nome, in particolare, il mapping visto nell'Esempio 15.3 è valido non soltanto per i tipi primitivi come interi e stringhe ma anche per tipi complessi come, per esempio, l'oggetto di classe `Person` mostrato nell'Esempio 15.4.

Esempio 15.4

```
public class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public int Age { get; set; }
}
```

La ricostruzione dei parametri, infatti, avviene sempre attraverso le stesse sorgenti dati, ma tramite l'uso di reflection e di ricorsione con mapping definito come `parameter_name.property_name`. Supponendo di avere l'Esempio 15.3 che accetta in ingresso un oggetto di tipo `Person`, questo avrà valore `null` fino a quando dall'URL (per chiamate HTTP `GET`) non verranno ricavate le proprietà corrispondenti come `FirstName`, `LastName` o `Age` (in modalità case-insensitive). Nel caso in cui ci sia anche una proprietà composta, come per esempio `Address`, in cui possono essere esplicitati città e via di residenza, il tutto verrà ricostruito allo stesso modo, purché il nome della proprietà corrisponda con quello esposto dalla sorgente. Nel caso particolare di liste e collezioni, il mapping avverrà tramite il modello `parameter_name[index]` (o, più semplicemente, tramite il solo [index] numerico), mentre nel caso dizionari si può specificare direttamente la chiave tramite [key]. Un URL che potrebbe rispettare questa convenzione ed essere in grado di ricostruire l'oggetto è, per esempio `https://www.miosito.com/home/index?firstName=mario&lastName=rossi`. Dobbiamo notare che in questo caso non si sta più sfruttando la route come sorgente da cui prelevare i valori, ma la querystring, e alla action del controller questo è del tutto trasparente perché continuerà a ricevere i valori che si aspetta.

Per evitare collisioni in quei casi in cui lo stesso parametro viene definito in più sorgenti, il framework mette a disposizione quattro attributi per specificare da dove recuperare il valore corretto: `FromHeader`, il cui binding viene fatto sugli header della richiesta HTTP, `FromQuery`, dalla querystring, `FromRoute`, dalla route definita nella classe `Startup` e `FromForm`, dalla FORM inviata. Tutti questi attributi possono essere utilizzati singolarmente oppure in combinazione fra di loro per recuperare valori provenienti da sorgenti differenti, come viene mostrato nell'Esempio 15.5.

Esempio 15.5

```
public ActionResult Index([FromHeader(Name = "Content-Type")] string ct,
   [FromRoute] string id)
{
   return View();
}
```

Come è dimostrato nell'Esempio 15.5, la lettura del `Content-Type` avviene tramite l'header il cui nome è corrispondente, mentre il binding del parametro `id` è rimasto pressoché invariato rispetto all'Esempio 15.3, se non per via della specifica della sorgente dati. Tutti gli attributi elencati, inoltre, dispongono anche di una proprietà `Name` tramite la quale si può specificare il nome del parametro che il model binder deve ricercare nella sorgente specificata: in questo modo si può creare un parametro all'interno della action con un nome diverso, rimuovendo di fatto il vincolo dei nomi previsto dal model binder. Oltre a questi attributi, ne esistono altri due che permettono alcune variazioni sul tema, come `FromBody`, che permette il recupero dei valori dal body della richiesta e sfrutta i formatter, come abbiamo già visto nel Capitolo 11, per ricostruire i valori dai formati opportuni come XML e JSON, e `FromServices`, che permette di iniettare i servizi tramite la dependency injection come parametri dell'action anziché a livello di costruttore, come viene fatto solitamente. Questo secondo attributo, in particolare, risulta utile laddove si ha bisogno di un servizio solo in una action specifica e in cui l'inizializzazione a livello di costruttore sarebbe troppo onerosa per tutte le altre chiamate.

Il mapping che effettua il model binder, come abbiamo detto nei paragrafi precedenti, avviene in modo ricorsivo su tutte le proprietà che vengono trovate, che siano rappresentate da tipi primitivi o da tipi composti. Nel caso in cui si voglia creare, per esempio, un nuovo oggetto all'interno di un database tramite una chiamata HTTP `POST`, però, non è detto che si abbiano a disposizione immediatamente tutti i parametri richiesti (come per esempio l'`id` per identificare in modo univoco quell'oggetto), per questo, anziché creare un nuovo modello *ad hoc*, può essere comodo applicare un filtro sulle proprietà da mettere in binding, come è mostrato nell'Esempio 15.6.

Esempio 15.6

```
public IActionResult Binding([Bind("FirstName,LastName")] Person p)
{
   // p.Age sarà sempre 0
   return View();
}
```

Nell'Esempio 15.6 si può notare come l'attributo `Bind`, che di default può essere omesso e sta a indicare che l'oggetto `Person` può essere recuperato da una sorgente qualsiasi, ha richiesto in modo esplicito il binding delle sole proprietà `FirstName` e `LastName` della classe. Pertanto, se nella richiesta verranno specificati altri parametri, come per esempio l'età, questi verranno completamente ignorati e ci si ritroverà con il solo valore di default (dipendente dal tipo). Qualora questo diventi il comportamento standard per ogni oggetto di tipo `Person`, anziché esplicitare singolarmente tutti i campi come stringhe – comportamento che è soggetto a potenziali errori, considerando che non c'è supporto per IntelliSense – è anche possibile intervenire in modo preciso sulle proprietà della classe stessa, come è dimostrato nell'esempio seguente.

Esempio 15.7

```
public class Person
{
   [BindRequired]
   public string FirstName { get; set; }
   [BindRequired]
   public string LastName { get; set; }
   [BindNever]
   public int Age { get; set; }
}
```

Come viene mostrato nell'Esempio 15.7, ci sono a disposizione due nuovi attributi, `BindRequired` che implica che il binding deve sempre essere effettuato dal model binder, e `BindNever`, che fa l'esatto opposto. Una volta specificati questi attributi, si potrà andare a rimuovere l'attributo `Bind` impostato nell'Esempio 15.6, poiché il comportamento sarà equivalente, a meno di eventuali altri campi che si vogliono mettere in mapping solo in determinate action. Allo stesso modo, se uno dei parametri, come per esempio l'`id`, viene sempre esposto tramite l'URL e la navigazione avviene tramite una route che prevede l'`id` stesso, è possibile andare ad aggiungere l'attributo `FromRoute` sopra la proprietà, come viene mostrato nell'Esempio 15.8.

Esempio 15.8

```
public class Person
{
   [FromRoute]
   public Guid Id { get; set; }
}
```

Nonostante tutti questi esempi, esistono comunque scenari in cui il funzionamento previsto out-of-the-box dal model binder non è sufficiente e perciò è richiesta un minimo di estensibilità del servizio stesso: un esempio lo possiamo creare sulla base delle WebAPI, che abbiamo trattato nel Capitolo 11. Infatti, abbiamo già visto come si usino i DTO/ViewModel come tecnica alternativa per scambiare oggetti con la view. Inoltre, usare l'attributo `BindRequiredAttribute` sulle entità non è propriamente una best practice, dato che si va a mischiare un attributo specifico per il caso d'uso dell'applicazione web con il codice di business che potrebbe essere usato anche in altri ambiti.

Creazione di un model binder personalizzato

La creazione di un nuovo oggetto di tipo `Person` potrebbe richiedere l'invio di dati sensibili al server, che potrebbero essere intercettati e manipolati. Il JSON Web Token, o JWT, è uno standard che definisce una modalità di trasmissione di informazioni sicura tra client e server a partire da un oggetto JSON: poiché il payload, ovvero l'insieme dei dati relativi alla persona, può essere firmato tramite HMAC o algoritmi RSA, al momento della ricezione sul server si può fare una verifica della firma per controllare che i dati non siano stati modificati durante il trasferimento. Utilizzando questo sistema, la action che si occupa della creazione dell'oggetto deve contenere una stringa (dato che il payload è in formato JSON) e non più l'oggetto `Person` che veniva ricostruito tramite model binder e `JsonFormatter`: non esiste un componente all'interno del model binder che riesca a capire la logica di business che abbiamo personalizzato per costruire

l'oggetto originale. Nonostante potrebbe non sembrare un problema, questo richiede che venga verificato il JWT e ricostruito l'oggetto originale, ovvero `Person`, ogni volta che viene invocata la action. Così facendo, andremmo a introdurre del codice per l'elaborazione che va oltre lo scopo della action stessa (e per ogni altra action che lo richiede), duplicando il codice e, nel caso peggiore, ripetendo lo stesso errore in parti differenti del codice.

Il caso ideale sarebbe quello di creare un posto centrale, in cui questa elaborazione viene fatta una sola volta per tutti quei componenti che ne hanno bisogno. Per questo è necessario andare a estendere il comportamento previsto dal model binder. Per creare un model binder personalizzato è sufficiente andare a creare una classe che implementi l'interfaccia `IModelBinder`, come viene mostrato nell'Esempio 15.9.

Esempio 15.9

```
public class PersonBinder : IModelBinder
{
   public async Task BindModelAsync(ModelBindingContext bindingContext)
   {
      if (bindingContext.ModelType != typeof(Person))
         return;
      var body = string.Empty;

      // recupero del body come stringa
      using (var bodyStream = new StreamReader(bindingContext.HttpContext. Request.Body))
      {
         body = await bodyStream.ReadToEndAsync();
      }
      // body vuoto
      if (string.IsNullOrEmpty(body))
         return;
      // cerco di recuperare il contenuto come json dal jwt ricevuto nel body
      var jsonPerson = JWTHelper.DecodeToken(body);
      if (!string.IsNullOrEmpty(jsonPerson))
      {
         // deserializzo e faccio validazione del modello
         var person = JsonConvert.DeserializeObject<Person>(jsonPerson);
         bindingContext.Result = ModelBindingResult.Success(person);
      }
   }
}
```

Una volta aggiunta l'interfaccia `IModelBinder`, verrà richiesta l'implementazione del metodo `BindModelAsync` per definire la logica che deve seguire il binder: prima di tutto deve essere recuperato il modello verso il quale si farà il mapping, per assicurarsi che sia di tipo `Person`. Quindi viene recuperato dal body il payload, ovvero il JWT token inviato in `POST`, che viene successivamente elaborato tramite una classe helper (disponibile nell'esempio allegato al capitolo) in modo che venga tradotto in un JSON contenente l'oggetto `Person` e, in caso in cui la deserializzazione da `string` abbia successo, verrà assegnata la proprietà `Result` dell'oggetto `ModelBindingContext`, che rappresenta l'oggetto che verrà visualizzato nella action che, come viene evidenziato nell'Esempio 15.10, avrà un parametro di tipo `Person`.

Esempio 15.10

```
[HttpPost]
public IActionResult Decode([ModelBinder(typeof(PersonBinder))] Person p)
{
   if (p == null)
      return BadRequest();
   return Ok();
}
```

Come è visibile nell'Esempio 15.10, il parametro della action `Decode` non è più la stringa contenente il JWT token ma l'oggetto `Person`, in modo che la funzione possa occuparsi solamente della sua logica, senza doversi preoccupare di come il dato viene recuperato ed elaborato. Per agganciare il binder creato nell'Esempio 15.9 alla action `Decode`, è necessario marcare il parametro di tipo `Person` con l'attributo `ModelBinder` in cui viene specificato il tipo del binder personalizzato. Per testare questa chiamata si può provare a inviare una richiesta HTTP `POST`, per esempio con Postman, in cui all'interno del body verrà mandato un token ricavato da questo URL: `http://aspit.co/bpc`. La chiamata dovrebbe andare a buon fine perché, come mostrato nell'URL, il token è valido, pertanto risponderà con uno status code 200 (OK).

È importante però sottolineare che, poiché dall'Esempio 15.9 si sta facendo a tutti gli effetti la ricostruzione di un oggetto, potrebbe diventare necessario eseguire la validazione prima di restituirlo tramite l'assegnazione a `Result`. Questa validazione potrebbe essere fatta di nuovo sulla base degli attributi, come per esempio `Required`, già visti all'interno del Capitolo 7, ma richiederebbe reflection,

oppure, poiché in questo caso la validazione sarebbe abbastanza semplice e richiesta solo in un caso specifico di inserimento, può anche essere eseguita in linea con un controllo condizionale sulle singole proprietà, come viene mostrato nell'Esempio 15.11.

Esempio 15.11

```
// deserializzo e faccio validazione del modello
var person = JsonConvert.DeserializeObject<Person>(jsonPerson);

if (string.IsNullOrEmpty(person.FirstName) ||
    string.IsNullOrEmpty(person.LastName))
{
   // aggiungo l'errore sul ModelState
   bindingContext.ModelState.AddModelError("person", "firstName and lastName cannot be null or
empty");
   return;
}
```

Come si può vedere nell'Esempio 15.11, infatti, tra le proprietà esposte dal `ModelBindingContext` c'è `ModelState`, che permette di specificare tutti gli errori che possono essere verificati poi lato action tramite la proprietà `ModelState.IsValid`, cosa che abbiamo già illustrato nel Capitolo 7. Una volta ricostruiti gli oggetti e verificata la validazione, poiché, di nuovo, questo binder potrebbe essere utilizzato in diversi punti dell'applicazione, potrebbe diventare scomodo specificarlo tutte le volte, per questo lo si può creare a partire da un provider di tipo `IModelBinderProvider`, come è mostrato nell'Esempio 15.12.

Esempio 15.12

```
public class PersonBinderProvider : IModelBinderProvider
{
   public IModelBinder GetBinder(ModelBinderProviderContext context)
   {
      if (context.Metadata.ModelType == typeof(Person))
         return new BinderTypeModelBinder(typeof(PersonBinder));
      return null;
   }
}
```

Nell'Esempio 15.12 viene implementato il metodo `GetBinder`, proveniente dall'interfaccia `IModelBinderProvider`, che consente di analizzare il contesto per capire se il binder creato, ovvero `PersonBinder`, può essere utilizzato. Nel caso in cui arrivi una request, saranno controllati tutti i provider con i binder di default previsti da ASP.NET Core e, quindi, verrà controllato `PersonBinderProvider`: qualora ci sia una corrispondenza tra il tipo per il quale il binder è registrato e il tipo del parametro che si vuole avere all'interno della action, allora sarà possibile creare una nuova istanza di quel binder tramite la classe `BinderTypeModelBinder`, altrimenti, ritornando il valore `null`, rimanderemo lo stesso controllo a un altro provider che verrà successivamente nella pipeline di esecuzione della request.

L'ultimo passaggio consiste proprio nella registrazione di questo provider che può essere fatta direttamente tramite le `MvcOptions` dell'extension method `AddMvc` in fase di startup nel metodo `ConfigureServices`, come è illustrato nell'Esempio 15.13.

Esempio 15.13

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc(options =>
   {
      options.ModelBinderProviders.Insert(0, new PersonBinderProvider());
   });
}
```

Nell'Esempio 15.13, il provider `PersonBinderProvider` è stato registrato per primo all'interno della collezione dei provider forniti dal framework, proprio perché, così facendo, qualora il nostro provider non sia in grado di fare l'elaborazione, per esempio dei tipi primitivi o di altri tipi che non siano `Person`, rimandiamo al framework l'incarico di trovare un'alternativa tra gli altri provider che ha a disposizione. Analizzando meglio la lista dei provider contenuta in `ModelBinderProviders`, possiamo trovare, infatti, tutti i provider che hanno permesso finora il binding "automatico", in cui non abbiamo mai dovuto specificare nulla, tra cui il `SimpleTypeModelBinderProvider`, che si occupa del binding per tipi primitivi, piuttosto che `ComplexTypeModelBinderProvider`, per i tipi complessi o, ancora, il `BodyModelBinderProvider` che si occupa di prelevare dati dal body e di formattarli correttamente secondo l'`InputFormatter` selezionato.

Model binding di interfacce

L'uso delle interfacce porta spesso innumerevoli benefici nella programmazione a oggetti, in quanto, rappresentando una sorta di contratto che le classi devono implementare, permette di avere una forte separazione tra diversi strati di logica. In particolare, nel caso di model binding, le interfacce possono aiutare su due aspetti: il primo è la separazione tra la logica di business e l'interfaccia grafica poiché potrebbero essere usate al posto del Model (o ViewModel), mentre il secondo, non meno importante, è che garantirebbe, grazie al concetto di ereditarietà, anche il riuso di tutti i metadati applicati alle proprietà, come, per esempio, gli attributi di validazione.

Il model binder, in realtà, prevede già alcuni meccanismi per lavorare con le interfacce: infatti è in grado di ricostruire collezioni di elementi come `IEnumerable<T>` e `Dictionary<T,K>`, e di iniettare istanze di oggetti tramite il motore di Dependency Injection, facendo uso dell'attributo `FromServices` che abbiamo visto in precedenza. Purtroppo, questo secondo meccanismo non consente di fare il mapping ricorsivo degli elementi facenti parte del modello, pertanto anche questa volta è necessario personalizzare il model binder per costruire il mapping di elementi a partire da interfacce che non rappresentino liste di oggetti. Come abbiamo mostrato nell'Esempio 15.14, si è voluto rendere generico il codice illustrato nell'Esempio 15.4, a cui sono stati aggiunti gli attributi di `Required` (per rendere obbligatori i campi di nome e cognome) e `Range` (per limitare i valori previsti nel campo età).

Esempio 15.14

```
public interface IPerson
{
   [Required]
   string FirstName { get; set; }
   [Required]
   string LastName { get; set; }
   [Range(18, 100)]
   int Age { get; set; }
}
```

La classe `Person`, visibile nell'Esempio 15.15, sarà quindi molto simile a quella generata nell'Esempio 15.4 ma, in aggiunta, ha l'implementazione dell'interfaccia `IPerson`, vista nell'esempio precedente, e un nuovo attributo di tipo `ModelMetadataType`.

Esempio 15.15

```
[ModelMetadataType(typeof(IPerson))]
public class Person : IPerson
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public int Age { get; set; }
}
```

L'implementazione dell'interfaccia garantisce la presenza di tutte le proprietà – e di eventuali metodi in essa contenuti – ma l'uso dell'attributo `ModelMetadataTypeAttribute` permette di rimandare sulle stesse proprietà della classe anche tutti i metadati esposti dall'interfaccia: questo consentirebbe, per esempio, di creare una classe `Customer` con le stesse proprietà e le stesse validazioni, oltre che proprietà e metodi aggiuntivi specifici del nuovo contesto. La action in grado di ricevere l'oggetto potrebbe essere simile a quella evidenziata nell'Esempio 15.16.

Esempio 15.16

```
[HttpPost]
public IActionResult InterfaceBinding(IPerson person)
{
   return View(model);
}
```

Come si può notare dall'esempio, non è necessario fare uso dell'attributo `FromServices`, questo perché andremo esplicitamente a fare il binding con un model binder provider personalizzato. Nonostante non si stia facendo uso dell'attributo, però, è comunque bene andare a specificare al motore di Dependency Injection che dovrà essere fatta la sostituzione, quando richiesta l'istanza, come viene mostrato nell'Esempio 15.17.

Esempio 15.17

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddTransient<IPerson, Person>();
   // registrazione di MVC ed altri servizi...
}
```

La parte di view, invece, poiché adesso può essere separata dal modello grazie all'astrazione, può essere modificata facendo uso dell'interfaccia.

Esempio 15.18

```
@model Capitolo15.Models.IPerson

<form asp-action="InterfaceBinding" asp-controller="Home">
   <div class="form-group">
      <label asp-for="FirstName" class="control-label"></label>
      <input asp-for="FirstName" class="form-control">
      <span asp-validation-for="FirstName" class="text-danger"></span>
   </div>
   <!-- implementazione degli altri campi -->
   <button type="submit">Submit</button>
</form>
```

Come abbiamo evidenziato negli Esempi 15.16 e 15.18, la parte di view e di controller sono ora referenziate solamente dall'interfaccia e l'unico componente che conosce la vera implementazione è l'insieme dei servizi. Provando a inviare la form dell'Esempio 15.18, otterremo un errore che metterà in evidenza come non esista un model binder in grado di ricostruire l'oggetto: è proprio quello che ci aspettavamo in partenza, pertanto è necessario andare a creare il binder personalizzato per risolvere il problema.

Esempio 15.19

```
public class InterfaceModelBinder : ComplexTypeModelBinder
{
   public InterfaceModelBinder(IDictionary<ModelMetadata, IModelBinder> propertyBinder) :
base(propertyBinder)
   {
   }
   protected override object CreateModel(ModelBindingContext bindingContext)
   {
      var modelInterface = bindingContext.ModelType;
      var model = bindingContext.HttpContext.RequestServices
                                    .GetService(modelInterface);
      return model;
   }
}
```

Come possiamo notare nell'Esempio 15.19, questa volta non siamo andati a implementare l'interfaccia `IModelBinder`: il meccanismo di binding ricorsivo di ASP.NET Core è già funzionante su tutte le proprietà grazie a binder esistenti nel framework, come abbiamo già visto, pertanto è sufficiente riutilizzarne uno, come `ComplexTypeModelBinder` che permetta di analizzare tutte le proprietà, anche quelle annidate. Quello che ci manca a tutti gli effetti è il vero e proprio il modello da mettere in binding, in quanto finora abbiamo solamente avuto accesso all'interfaccia. Per farlo è necessario andare a implementare l'override del metodo `CreateModel`: con la proprietà `ModelType` esposta dal contesto di binding avremo accesso all'interfaccia e, grazie alla proprietà `RequestServices` potremo recuperare tramite il motore di DI il modello corrispondente all'interfaccia stessa, ovvero la classe `Person`.

Come abbiamo già mostrato in precedenza, a questo punto bisogna registrare un provider personalizzato per permettere al binder `InterfaceModelBinder` di lavorare. Come abbiamo già discusso ed evidenzieremo nuovamente nell'esempio seguente, come prima cosa è bene andare a controllare l'esistenza del contesto di binding, mentre, in secondo luogo, andiamo a controllare se il binder che vogliamo aggiungere è in grado di rispondere all'esigenza che abbiamo previsto, quindi al binding per le interfacce.

Esempio 15.20

```
public class InterfaceModelBinderProvider : IModelBinderProvider
{
   public IModelBinder GetBinder(ModelBinderProviderContext context)
   {
      if (context == null)
         throw new ArgumentNullException(nameof(context));
      if (context.Metadata.ModelType.GetTypeInfo().IsInterface &&
            !context.Metadata.IsCollectionType &&
            !context.BindingInfo.BindingSource
                        .CanAcceptDataFrom(BindingSource.Services))
      {
         var binders = context.Metadata.Properties
                                    .ToDictionary(p => p, p => context.CreateBinder(p));
         return new InterfacesModelBinder(binders);
      }
      return null;
```

```
    }
}
```

In particolare, nell'Esempio 15.20 i parametri per cui il binder verrà invocato sono tre: l'istanza di binding deve essere una interfaccia, ma non deve essere una collezione (dato che ci sono binder dedicati) e non deve essere marcata dall'attributo `FromServices`. Una volta specificate le condizioni, può essere creato il dizionario da passare al costruttore di `InterfaceModelBinder` visto nell'Esempio 15.19: per ogni proprietà esposta all'interno del modello `Person`, infatti, viene creata una nuova istanza del binder (per esempio `SimpleTypeModelBinder` per tipi primitivi come interi, date e stringhe, oppure `ComplexTypeModelBinder` per tipi complessi come `Address` per indicare un eventuale indirizzo) e quindi viene creato il binder in grado di ricostruire l'interfaccia.

Esempio 15.21

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddTransient<IPerson, Person>();
   services.AddMvc(options =>
   {
      options.ModelBinderProviders.Insert(0,
                        new InterfaceModelBinderProvider());
   });
}
```

Come abbiamo già visto in precedenza e ripreso dall'Esempio 15.21, è necessario andare a registrare il binder provider appena creato, tenendo conto dell'ordinamento: poiché a priori non sappiamo se verrà richiesto dal provider, con la chiamata a `CreateBinder`, un provider di default o uno personalizzato come `PersonBinder` costruito nell'Esempio 15.9, è bene fare in modo che sia il primo a essere inserito nell'elenco dei provider, così che il framework li controlli tutti in sequenza per determinare il più adatto. Nonostante quello che è stato approfondito all'interno di questo paragrafo, stiamo ancora dando per scontato che le interfacce siano presenti e accessibili all'interno del progetto stesso ma, purtroppo, non è sempre così. Pertanto diventa fondamentale capire se è comunque possibile sfruttare lo stesso livello di estendibilità del framework.

Modifica dei metadati

ASP.NET Core offre un ottimo supporto per tutta la parte relativa agli attributi, sia per quanto riguarda DataAnnotation sia per quanto riguarda altri metadati utili per la validazione e il binding, come abbiamo già avuto modo di vedere anche nei capitoli precedenti. Può succedere però che, come nel caso delle interfacce, spesso la definizione del modello arrivi da parti del codice non accessibili o estendibili come, per esempio, librerie aggiunte al progetto. Per questo, talvolta, è necessario modificare gli attributi per poterli rendere adeguati al loro contesto di utilizzo, in modo che non siano più considerati generici come esposti dalla libreria.

Per farlo è sufficiente estendere il comportamento previsto da `IMetadataDetailsProvider` secondo lo scenario che più ci interessa. Supponiamo, per esempio, di voler andare a impostare l'attributo `DisplayName` per tutte quelle proprietà che non lo dispongono: l'Esempio 15.22 dimostra come impostare il valore mettendo la prima lettera maiuscola a ogni proprietà.

Esempio 15.22

```
public class MetadataDetailsProviderCustom : IDisplayMetadataProvider
{
   public void CreateDisplayMetadata(DisplayMetadataProviderContext context)
   {
      var propertyAttributes = context.Attributes;
      var modelMetadata = context.DisplayMetadata;
      var propertyName = context.Key.Name;
      if (!propertyAttributes.OfType<DisplayAttribute>().Any())
         modelMetadata.DisplayName = () => char.ToUpper(propertyName[0]) +
                                                propertyName.Substring(1);
   }
}
```

Nell'esempio siamo andati a creare una classe `MetadataDetailsProviderCustom` in cui abbiamo implementato il metodo `CreateDisplayMetadata` proveniente dall'interfaccia `IDisplayMetadataProvider`: questa ci permette quindi di accedere a tutto il contenuto riguardante gli attributi `Display` e ci consente di verificare su quali proprietà verrà applicato. In modo molto simile a quello evidenziato nell'Esempio 15.22, sarà possibile, per esempio, registrarsi per l'uso di Humanizer, una libreria che permette di aggiungere gli spazi tra le proprietà che hanno nomi composti, in modo che l'attributo `DisplayName` sia più leggibile a un umano, piuttosto che effettuare la traduzione andando a recuperare una implementazione di `IStringLocalizer`, come vedremo nel capitolo successivo.

Nel caso in cui, invece, volessimo andare a modificare gli attributi di validazione, come mostrato dall'Esempio 15.23, possiamo implementare l'interfaccia `IValidationMetadataProvider` che ci dà accesso al contesto di validazione tramite il metodo `CreateValidationMetadata`.

Esempio 15.23

```
public class MetadataDetailsProviderCustom : IValidationMetadataProvider
{
   public void CreateValidationMetadata(ValidationMetadataProviderContext context)
   {
      var propertyAttributes = context.Attributes;
      var modelMetadata = context.ValidationMetadata;
      var propertyName = context.Key.Name;
      if (propertyAttributes.OfType<RangeAttribute>().Any())
      {
         foreach (var attribute in propertyAttributes.OfType<RangeAttribute>())
         {
            attribute.ErrorMessage = $"La proprietà {propertyName} deve rispettare il range!";
         }
      }
   }
}
```

Nell'Esempio 15.23, in particolare, siamo andati a recuperare tutti gli attributi di tipo `RangeAttribute` (come, per esempio, quello impostato per il campo `Age` della classe `Person`, definita nell'Esempio 15.15) per applicare una modifica al messaggio di errore che dovrà essere mostrato quando il valore inserito dall'utente non sarà contenuto tra il minimo e il massimo impostato dall'attributo stesso.

Allo stesso modo, è possibile andare a cambiare i meccanismi di binding, semplicemente implementando l'interfaccia `IBindingMetadataProvider`, come mostrato nell'esempio seguente.

Esempio 15.24

```
public class MetadataDetailsProviderCustom : IBindingMetadataProvider
{
   public void CreateBindingMetadata(BindingMetadataProviderContext context)
   {
      if (context.PropertyAttributes != null &&
            context.PropertyAttributes.OfType<RequiredAttribute>().Any())
      {
         context.BindingMetadata.IsBindingRequired = true;
      }
   }
}
```

Nell'Esempio 15.24 viene dimostrato come applicare l'attributo `BindRequired` su tutte le proprietà che hanno già impostato l'attributo `Required`. In questo caso è necessario controllare che siano presenti degli attributi, poiché il binding potrebbe essere fatto troppo presto perché essi siano presenti. Una volta modificata la classe che gestisce i vari metadata provider di cui abbiamo bisogno, sarà necessario registrarla all'interno delle `MvcOptions` durante la configurazione dei servizi, per fare in modo che venga recuperata dal framework, come viene mostrato nell'Esempio 15.25.

Esempio 15.25

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc(options =>
   {
      options.ModelMetadataDetailsProviders.Insert(0,
            new MetadataDetailsProviderCustom());
   });
}
```

Esattamente come abbiamo già esaminato per i model binder provider, anche per il `ModelMetadataDetailsProvider` c'è un ordine che va rispettato. Per questo è bene registrare gli elementi personalizzati prima di tutti quelli previsti dal framework. Tutto il codice visto finora viene trasformato in automatico in codice HTML che il browser riesce a interpretare, ma non è sempre così: ci sono casi infatti in cui vogliamo che il markup venga generato a partire dal server: per questo abbiamo già introdotto il tema dei tag helper e ne approfondiremo ora il loro utilizzo.

Sviluppare un tag helper personalizzato

Nel Capitolo 6 abbiamo visto qual è la differenza tra HTML helper e tag helper in termini di utilizzo e abbiamo visto come sfruttare quest'ultimi per sposare una sintassi basata su markup, ma che allo stesso tempo sappia arricchire elementi standard di HTML5 oppure ne permetta la creazione di nuovi. In entrambi i casi, i tag helper sono dei generatori di markup interamente implementati tramite classi scritte appositamente che, eventualmente, possono mischiare o manipolare quanto abbiamo scritto nelle view Razor.

Abbiamo già visto che nel file `_ViewImports.cshtml` possiamo inserire del codice condiviso tra tutte le view e normalmente troviamo sempre la direttiva dell'Esempio 15.26.

Esempio 15.26

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Essa indica al parser di Razor di caricare tutte le classi di tipologia tag helper presenti nell'assembly, al fine di riconoscere gli elementi e gli attributi speciali che necessitano di un processamento. Queste classi sono identificate dal fatto che ereditano da TagHelper del namespace `Microsoft.AspNetCore.Razor.TagHelpers`, perciò anche noi possiamo fare altrettanto, preparando una classe, come viene mostrato nell'Esempio 15.27.

Esempio 15.27

```
[HtmlTargetElement("if-time", TagStructure = TagStructure.NormalOrSelfClosing)]
public class ConditionalTimeTagHelper : TagHelper
{
   public override void Process(
      TagHelperContext context,
      TagHelperOutput output)
   {
   }
}
```

La classe è adornata con l'attributo `HtmlTargetElement` che ci permette di specificare il nome del tag da utilizzare nella view, che per noi sarà `<if-time />`. Questa informazione è facoltativa perché Razor deduce il nome del tag in base al nome della classe secondo la nomenclatura in stile lower-kebab case. In pratica, senza l'attributo dovremmo utilizzare il tag `<conditional-time />` di minore facilità nell'utilizzo, perciò sfruttiamo l'attributo, anche per indicare se il nostro tag può avere un tag di chiusura o no. Nel nostro esempio vogliamo realizzare un tag helper che ci permetta di condizionare la visualizzazione di un pezzo di markup in base all'orario, perciò vogliamo che abbia un tag di chiusura per consentire l'inserimento di un contenuto figlio.

Proseguendo l'analisi dell'Esempio 15.27, possiamo vedere il cuore implementativo concentrato nel metodo `Process` del quale dobbiamo fare l'override. Riceve il `TagHelperContext` contenente gli attributi e le informazioni inerenti al tag stesso e, cosa ancora più importante, riceve un `TagHelperOutput` che ci permette di controllare cosa mandare in uscita. Per sfruttare il tag helper, che per il momento non fa nulla, non ci resta che aggiungere una nuova riga al file `_ViewImports.cshtml` con il nome dell'assembly che contiene la classe, come abbiamo fatto nell'Esempio 15.26. Nella view possiamo di conseguenza inserire il nostro tag, di nome `if-time`.

Esempio 15.28

```
<if-time>
   <h1>Contenuto giornaliero</h1>
</if-time>
```

La pagina, se eseguita, genera un HTML simile a quello dell'Esempio 15.28, perché normalmente un tag helper renderizza sé stesso e il proprio contenuto. Interveniamo quindi nel metodo `Process` per escludere l'emissione del tag, non valido nello standard HTML5, e inoltre inseriamo la logica per inibire la generazione dell'output nel caso l'orario attuale non lo consenta.

Esempio 15.29

```
// Evito di scrivere il tag stesso
output.TagName = null;

int hour = DateTime.Now.Hour;
if (hour > 20)
{
   // Non emetto per intero l'output
   output.SuppressOutput();
}
```

L'oggetto `TagHelperOutput` ci consente di rendere nullo il `TagName` o, nel caso, controllare il `TagMode` per scegliere lo stile di chiusura del tag. Oltre a questo, grazie al metodo `SuppressOutput` possiamo chiedere di omettere per intero l'output. Utilizzare orari fissi non rende però comodo l'utilizzo del componente, perciò procediamo inserendo delle proprietà.

Esporre proprietà sul tag helper

Dato che un tag helper è rappresentato da una normale classe, possiamo aggiungere su di essa anche delle proprietà, per indicare l'intervallo orario per regolare la visibilità del contenuto, come viene mostrato nell'Esempio 15.30.

Esempio 15.30

```
public class ConditionalTimeTagHelper : TagHelper
{
   public int MinHour { get; set; } = 8;
   [HtmlAttributeName("max-hour")]
   public int MaxHour { get; set; } = 20;
   public override void Process(
      TagHelperContext context,
      TagHelperOutput output)
   {
      int hour = DateTime.Now.Hour;
      if (hour < MinHour || hour > MaxHour)
      {
         // ...
```

L'uso dell'attributo `HtmlAttributeName`, anche in questo caso, è facoltativo. Le proprietà seguono la stessa convenzione sulla nomenclatura, poiché posso essere utilizzate come attributi sul tag stesso. La Figura 15.1 mostra come in fase di utilizzo nella view, l'IntelliSense di Visual Studio riconosca automaticamente il tag helper e, successivamente, gli attributi.

Figura 15.1 – Supporto ai tag helper e agli attributi da parte di Visual Studio.

L'assegnazione delle proprietà viene validata direttamente in fase di compilazione perciò non possiamo erroneamente assegnare una stringa a un intero, come nel nostro caso. Inoltre, possiamo assegnare agli attributi espressioni C# tramite la keyword @ di Razor, per recuperare valori da variabili o dal model, come mostrato nell'Esempio 15.31.

Esempio 15.31

```
@{
   var interval = new { min = 10, max = 20};
}
<if-time min-hour="@interval.min" max-hour="@interval.max">
   <h1>Contenuto giornaliero</h1>
</if-time>
```

Oltre a proprietà classiche che possono contenere tipi primitivi ma anche tipi complessi, possiamo sfruttare alcuni oggetti speciali. Il primo è di tipo `ViewContext`, la cui omonima proprietà è disponibile nelle view e nelle partial view, ma che possiamo sfruttare anche all'interno del tag helper. Con essa abbiamo accesso all'`HttpContext`, al `ViewData` e ai metadata del modello disponibile nel contesto della view. È sufficiente inserire una proprietà di questo tipo marcata con l'attributo omonimo, come è mostrato nell'Esempio 15.32, per ottenere l'oggetto valorizzato automaticamente a runtime.

Esempio 15.32

```
[HtmlAttributeNotBound]
[ViewContext]
public ViewContext ViewContext { get; set; }
```

L'attributo `ViewContext` è necessario, mentre la proprietà non deve necessariamente chiamarsi con lo stesso nome. L'uso del secondo attributo, di nome `HtmlAttributeNotBound`, è necessario per evitare che l'IntelliSense e il compilatore permettano di valorizzare la proprietà `ViewContext` come attributo. Non è detto, infatti, che ogni proprietà aggiunta nella classe debba essere raggiungibile come attributo. Possiamo vedere nella Figura 15.2 come in fase di processamento troviamo nell'oggetto molte informazioni preziose sul contesto.

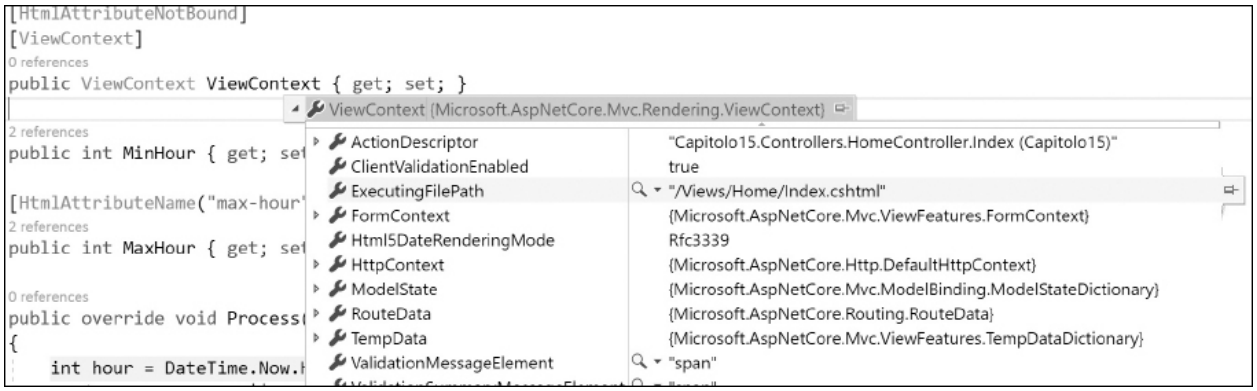


Figura 15.2 – Informazioni presenti nel ViewContext utili per il tag helper.

Una seconda tipologia di oggetto è rappresentata dal tipo ModelExpression, che ci permette di implementare le stesse logiche offerte dai tag helper per `<input />` o `<span />`. Possiamo indicare in Razor un'espressione attinente al modello della view in cui il tag helper viene utilizzato. Aggiungiamo una nuova proprietà alla classe, come mostrato nell'Esempio 15.33.

Esempio 15.33

```
[HtmlAttributeName("time-for")]
public ModelExpression For { get; set; }
```

Nome della proprietà e attributo sono facoltativi, ma ci richiamano un comportamento già noto. Possiamo, infatti, sfruttare l'IntelliSense per valorizzare l'attributo `time-for` con i membri del modello della view, come viene mostrato nella Figura 15.3.

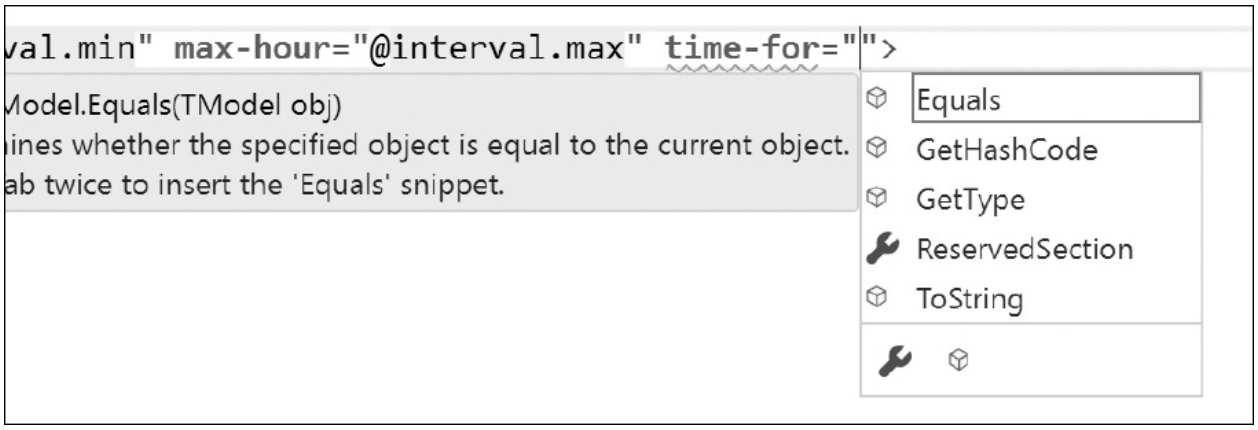


Figura 15.3 – Selezione di una proprietà del modello della view tramite Visual Studio.

Otteniamo quindi un controllo a compile-time e, soprattutto, un oggetto che espone l'espressione utilizzata ma anche altre proprietà utili, quali:

- `Model`: restituisce il valore della proprietà del modello.
- `Metadata`: espone tutte le informazioni risolte da ASP.NET sulla proprietà sulla base degli attributi di visualizzazione e di validazione.
- `ModelExplorer`: permette di navigare tra i metadati del contenitore o di proprietà figlie del modello stesso.

Tutte queste informazioni sono fondamentali qualora vogliamo emettere attributi o testo sulla base della proprietà, allo stesso modo di come fanno i tag helper nativi, che valorizzano attributi come `id`, `name` ecc. A questo punto, cerchiamo di entrare più nel dettaglio di cosa possiamo fare dal punto di vista della generazione del contenuto.

Generare HTML nei tag helper

L'oggetto `TagHelperOutput` che riceviamo sul metodo `Process` contiene quanto necessario per manipolare il tag, i suoi attributi e il suo contenuto. Nell'Esempio 15.34 possiamo vedere una serie di istruzioni che danno l'idea di ciò che possiamo fare.

Esempio 15.34

```
// Imposto il tag
output.TagName = ”div”;
// Imposto la chiusura del tag
output.TagMode = TagMode.StartTagAndEndTag;
// Aggiungo un attributo
output.Attributes.Add(”id”, For.Name);

// Sovrascrivo il contenuto
output.Content.SetContent(”Non consentito in questo momento”);

// Aggiungo il contenuto prima e dopo il tag
output.PreElement.SetContent(”pre <div>”);
output.PostElement.SetContent(”post <div>”);

// Aggiungo il contenuto prima e dopo il contenuto stesso
output.PreContent.SetContent(”<div> pre”);
output.PreContent.SetContent(”<div> post”);
```

L’utilizzo è piuttosto intuitivo e dà un accesso mirato al contenuto o addirittura all’esterno del tag stesso. Qualora la generazione dell’HTML si facesse più complessa, è caldamente sconsigliato concatenare stringhe, creare tag e includere stringhe provenienti dall’utente, poiché dovremmo occuparci di tutto l’encoding necessario. Per evitare problemi di sicurezza ed essere più veloci nella scrittura di HTML, l’oggetto `TagBuilder` ci permette facilmente di generare porzioni di markup, come mostrato nell’Esempio 15.35.

Esempio 15.35

```
if (hour < MinHour || hour > MaxHour)
{
   // Preparo lo <span>
   var span = new TagBuilder(”span”);
   span.Attributes.Add(”min”, MinHour.ToString());
   span.InnerHtml.Append(”Accesso non consentito”);
   output.Content.SetHtmlContent(span);
}
```

L’oggetto va creato specificando il tag che vogliamo rappresentare e dispone di metodi per popolare gli attributi, gli stili e valorizzare il suo contenuto. Fatto questo, poiché implementa l’interfaccia `IHtmlContent`, possiamo associare l’istanza di `TagBuilder` al contenuto dell’output tramite la chiamata `SetHtmlContent`. In questo modo generiamo in maniera sicura e rapida il nostro markup e rendiamo più leggibile il nostro codice.

Conclusioni

Nel corso di questo capitolo abbiamo visto nuovamente come estendere il comportamento previsto di default da ASP.NET Core, discutendo nel dettaglio di cosa avviene nella fase immediatamente successiva alla ricezione della request HTTP, per capire come i dati provenienti da querystring, body, header e altre sorgenti vengono recuperati e ricostruiti per essere poi iniettati come oggetti all’interno di una action contenuta nel controller. Dall’altro lato, abbiamo discusso dei tag helper, ovvero di uno strumento che ci consente di scrivere tag HTML a partire dal server, analizzando, di fatto, quello che succede nell’istante immediatamente precedente l’emissione dell’output e del render in HTML del codice.

Nel prossimo capitolo approfondiremo un altro tema legato all’estendibilità, anche se non si discuterà più del framework in sé, ma di come il framework ci permetta di essere raggiungibili e accessibili grazie al supporto di localizzazione e globalizzazione.

16
Supporto alla globalizzazione e internazionalizzazione

La creazione di un sito web o di un'applicazione in genere che ha necessità di diffondersi a livello mondiale deve poter essere non solo raggiungibile ma anche accessibile a chiunque. Con accessibilità però, non s'intende esclusivamente il supporto per persone che hanno, per esempio, disturbi visivi ma, più in generale, accessibilità significa rendere il servizio che si sta offrendo fruibile con facilità a qualsiasi tipologia di utente finale.

Volendo esporre un caso più concreto, l'inglese è sicuramente la lingua più conosciuta e parlata nel mondo occidentale ma non è detto che sia fruibile da parte di tutti gli utenti, quindi, localizzare i contenuti, per esempio, in italiano, rende il sito stesso accessibile a tutti gli utenti italiani. Un concetto un po' diverso ma strettamente correlato, riguarda invece la visualizzazione di date o di valute, in cui sia il formato sia l'importo possono differire per culture e regioni.

All'interno di questo capitolo, affronteremo come primo tema un'introduzione a quelli che sono i principi e la terminologia che circonda il mondo dell'internazionalizzazione e quindi parleremo di tutti quelli che sono gli strumenti che .NET Core e, in particolare ASP.NET Core, mettono a disposizione per creare applicazioni accessibili. Per riprendere quello che è già stato affrontato nei capitoli precedenti, parleremo anche di come l'internazionalizzazione viene applicata non solo ai contenuti e, tramite Razor, alle view, ma anche di come possiamo localizzare con data annotation e di come possiamo sfruttare i middleware per personalizzarne l'utilizzo.

I requisiti minimi

Per includere il supporto alla localizzazione nelle applicazioni ASP.NET Core, il componente minimo da includere è il pacchetto di NuGet `Microsoft.AspNetCore.Localization`. Ci sono anche altri pacchetti opzionali, per raggiungere un supporto completo alla localizzazione, come:

- `Microsoft.Extensions.Localization`: include la localizzazione, supporto a file di risorse e opzioni per la personalizzazione.
- `Microsoft.AspNetCore.Localization.Routing`: include la localizzazione a livello di routing (l'esempio classico è la specifica della localizzazione in query string come `miosito.com/it-IT/Home/Index`).
- `Microsoft.AspNetCore.Mvc.Localization`: include la localizzazione per tutti i componenti di MVC, compresa la localizzazione delle view.
- `Microsoft.AspNetCore.Mvc.DataAnnotations`: include la localizzazione per tutti gli attributi di validazione e delle data annotation.

Per le applicazioni ASP.NET Core 2.0, non è necessario installare tutti questi pacchetti poiché sono già inclusi all'interno del meta-pacchetto `Microsoft.AspNetCore.All` e pertanto iniziare sarà molto semplice.

Internazionalizzazione, globalizzazione e localizzazione

Quando si parla di internazionalizzazione, in realtà si confondono (o si fa riferimento a loro in modo indifferente) diversi termini: globalizzazione, localizzazione e internazionalizzazione. Questi tre termini sono strettamente correlati ma includono significati diversi poiché il loro utilizzo è differente. L'internazionalizzazione è la parte più semplice, infatti è l'unione di localizzazione e globalizzazione. La globalizzazione è, in linea di massima, il primo passo che viene eseguito in fase di design dell'applicazione stessa, perché è qui che si impostano le fondamenta che verranno utilizzate poi nei successivi processi di localizzazione. Probabilmente abbiamo già sentito nominare la globalizzazione con un altro termine: "G11n". Questo termine è solamente una abbreviazione in cui il numero "11" è il numero di lettere che separa i caratteri iniziali e finali di "Globalization".

In questo passaggio non scriviamo codice né iniziamo a fare la traduzione dei contenuti. È solo una fase di design, ed è qui che si prendono decisioni riguardanti:

- Le lingue che devono essere supportate.
- La tipologia di supporto: si potrebbe cambiare lingua tramite route specifiche piuttosto che tramite cookie o tramite altri sistemi personalizzati.
- I valori che devono essere utilizzati come placeholder per i valori reali: in questa categoria rientrano i valori che devono essere tradotti in una lingua specifica o per formati diversi di visualizzazione (date piuttosto che valute).

- I formati che devono essere supportati: parlando di valute, possiamo decidere di utilizzare solo l'euro, nonostante l'applicazione possa essere localizzata in inglese e quindi manchi di supporto per la sterlina o per il dollaro.

- La culture di default: in caso un utente provi ad accedere, per vie più o meno supportate e pensate dall'applicazione, a contenuti che non sono disponibili in una lingua o in un determinato formato.

La localizzazione, invece, è un processo che avviene una volta terminata la fase di design e che, per via di com'è costruito ASP.NET Core, ovvero per via della dependency injection, può essere fatta anche una volta che l'applicazione stessa è già nell'ambiente di produzione. Anche in questo caso, la localizzazione ha una sua abbreviazione, "L10n", in cui 10 è ancora una volta il numero di lettere che separa le iniziali e finali di "Localization". Questo passaggio, se intendiamo la sola traduzione letterale nelle culture che abbiamo scelto in fase di progettazione, è teoricamente molto semplice e l'unico sforzo necessario è quello di trovare persone adatte a eseguire la traduzione e scrivere un paio di righe di codice. Questo principio di sola traduzione letterale può anche essere trovato abbreviato come "L12y". Nel caso in cui, invece, si voglia parlare integralmente di localizzazione, bisogna prestare molta attenzione al contenuto che deve essere mostrato: una data in formato americano mostra il mese prima del giorno e, quindi, se cercassimo di applicare una conversione di una data in formato europeo al formato americano, potremmo incorrere in errori a runtime. Proprio per questo la complessità varia in base alle scelte fatte nel processo di globalizzazione e, qualsiasi siano le nostre scelte, sarà opportuno applicare i vari test per verificare che tutto funzioni.

Lavorare con le culture

Per occuparsi di internazionalizzazione, bisogna interagire con le differenti culture. Ogni culture, o locale, in .NET Core, è rappresentata da una serie di regole e formati specifici sia per lingua sia per area geografica e, in generale, lo sviluppatore non deve preoccuparsi di tutte le regole, poiché saranno il framework e il sistema operativo a fornire una implementazione di base.

Ogni culture supportata da .NET Core, viene rappresentata e referenziata tramite quelli che sono chiamati "language tag", ovvero delle abbreviazioni che permettono di individuare in modo univoco parametri come la lingua e l'area geografica. Questi tag sono ben definiti tramite lo standard RFC 5646 creato e gestito dall'Internet Engineering Task Force (IETF), ovvero lo stesso ente che si occupa di gestire i più grandi standard relativi a internet come OAuth e WebSocket. Un tag è piuttosto semplice e si compone solitamente di tre parti: la prima parte riguarda un identificativo della lingua abbreviato in due lettere in formato ISO 639, la seconda rappresenta l'eventuale dialetto mentre la terza indica, in formato ISO3166-1 sempre abbreviato in due lettere, l'area geografica di riferimento. Per identificare l'italiano, per esempio, si possono trovare diverse varianti, come "it-CH" che indica la lingua italiana ma con area geografica Svizzera (con i suoi formati), oppure "it-IT" o "it" che sottintendono la lingua italiana parlata in Italia in modo indifferente. Poiché non abbiamo varianti dialettali, per l'italiano non si trovano tag composti da tutte e tre le parti, ma un esempio potrebbe essere fatto con "sr-Latn-BA" che rappresenta la lingua serba parlata in Bosnia-Herzegovina con dialetto latino, piuttosto che "sr-Cyrl-BA" che rappresenta la stessa lingua e la stessa area geografica ma il dialetto di riferimento è questa volta il cirillico.

.NET Core, esattamente come il .NET Framework, include tutti questi concetti in una classe chiamata `CultureInfo`, mentre tutti gli elementi legati ai concetti di internazionalizzazione sono contenuti in classi dell'assembly `System.Globalization`. La classe `CultureInfo` è un po' cambiata rispetto al .NET Framework classico ma, nonostante non esista più la proprietà `Thread.CurrentThread`, non ne viene alterato il funzionamento, ovvero ogni culture verrà impostata e mantenuta per ogni singolo thread creato e, da .NET Core nello specifico, questo è vero anche per tutto il processo di routing fino a quando la risposta viene consegnata al client. Finora abbiamo parlato solamente di culture, ma analizzando la classe `CultureInfo` troviamo esposte due proprietà differenti:

- `CurrentCulture`: indica come devono essere analizzati date, formati, numeri e valute, ordinamenti e convenzioni per i confronti.

- `CurrentUICulture`: rappresenta ciò che deve essere mostrato nell'interfaccia grafica e riguarda i contenuti come testo e, come vedremo in seguito, HTML.

Poiché queste due culture sono esposte in due proprietà ben separate, non ci sono obblighi di avere gli stessi valori per le stesse proprietà: per esempio, supportare la lingua inglese non implica per forza cose di supportare il formato delle date americano, ma questo dipende molto dalla tipologia di applicazione, dalla logica di business e da quanto è definito nel processo di design. Al contrario, decidere di supportare un differente formato, per esempio di valute, può aggiungere una determinata complessità all'applicazione stessa: immaginando un sito di e-commerce, i prezzi, che prima dell'aggiunta dell'internazionalizzazione sono stati salvati su un solo campo di un database, non saranno identici sia in dollari sia per euro e sterline, perché non basta cambiare il simbolo, ma bisogna eventualmente applicare una conversione di valuta e capire il formato specifico (il separatore delle migliaia è diverso in base all'area geografica).

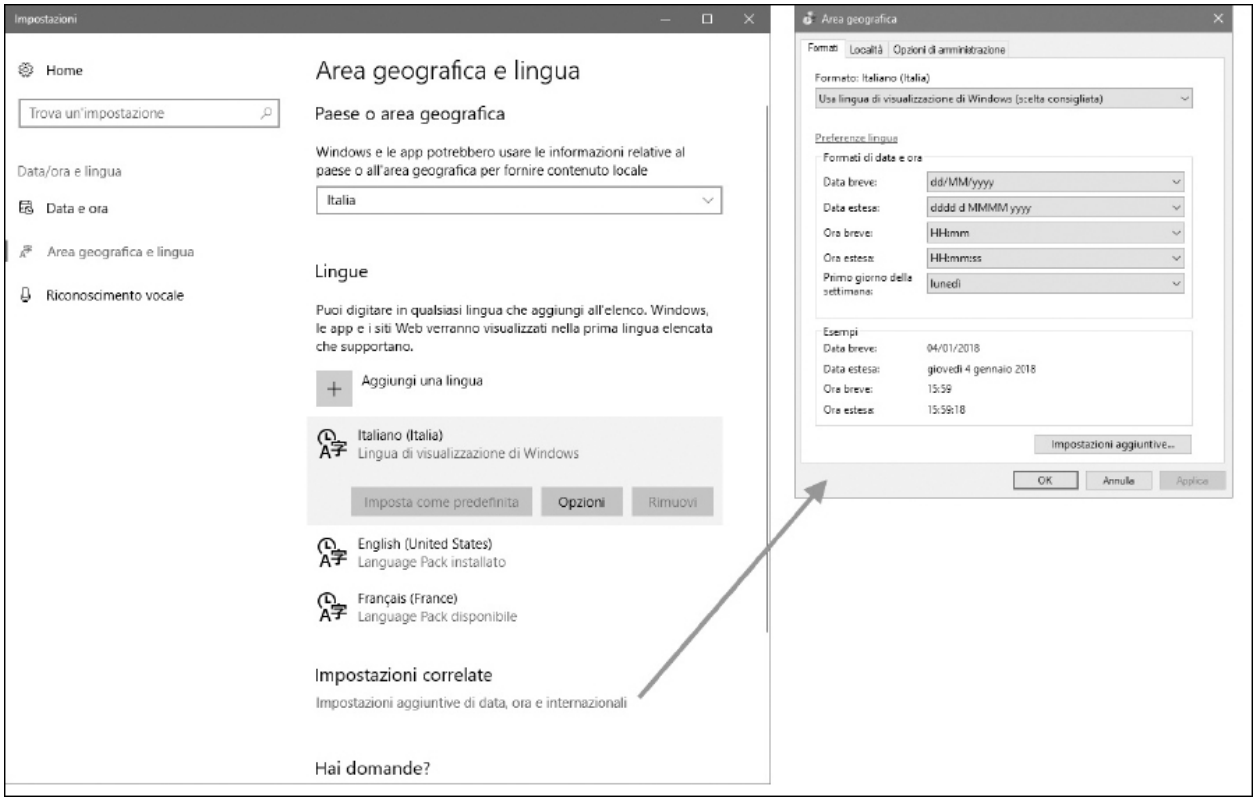


Figura 16.1 – Nelle impostazioni di Windows relative all'area geografica e alla lingua è possibile cambiare i valori corrispondenti alle `CurrentCulture` e `CurrentUICulture` di (ASP).NET Core.

Come viene evidenziato nella Figura 16.1, la `CurrentCulture` e la `CurrentUICulture` sono impostabili direttamente dal sistema operativo tramite gli appositi menù delle impostazioni di "Area geografica e lingua": il primo valore selezionato dall'elenco sarà il valore di default. Da codice, invece, è sufficiente assegnare il language tag all'apposita proprietà, come viene illustrato nell'Esempio 16.1.

Esempio 16.1

```
static void Main(string[] args)
{
   CultureInfo italian = new CultureInfo("it-IT");
   CultureInfo.CurrentCulture = italian;
   CultureInfo.CurrentUICulture = italian;
   Console.WriteLine(CultureInfo.CurrentCulture.Name);
   Console.WriteLine(CultureInfo.CurrentCulture.DisplayName);
}
```

L'Esempio 16.1, seppur riferito a un'applicazione console, mostrerà in output il nome della culture scelto, ovvero "it-IT" e il suo nome completo, ovvero "Italiano (Italia)", che è lo stesso valore visibile nelle impostazioni di Windows ed è coerente con quanto appena assegnato, ovvero lingua italiana parlata in Italia.

Per riprendere l'esempio fatto in precedenza riguardo ai numeri e, più nello specifico, alle valute, vediamo come l'Esempio 16.2, nonostante la sua semplicità, possa essere soggetto a errori quasi impensabili.

Esempio 16.2

```
void ChangeCulture()
{
   CultureInfo.CurrentCulture = new CultureInfo("en-US");
   NumbersDemo();
   CultureInfo.CurrentCulture = new CultureInfo("it-IT");
   NumbersDemo();
}

void NumbersDemo()
{
   string numberAsString = "10,500";
   decimal number = decimal.Parse(numberAsString) + 1;
```

```
   Console.WriteLine(number.ToString("C"));
}
```

L'Esempio 16.2 stamperà sulla console due valori, uno per culture, corrispondenti al valore "10,500" incrementato di uno e con il valore della valuta corrispondente alla culture. L'operazione di somma è piuttosto banale, ma i risultati ottenuti sono contrastanti: nel caso in cui la culture selezionata sia quella inglese, l'output generato sarà 10501$, mentre, quando la culture cambia in italiano, l'output diventa 11,50€: il carattere virgola contenuto nella variabile `numberAsString` assume un valore diverso secondo la culture impostata da separatore delle migliaia a separatore delle unità. È evidente che, quando sviluppiamo applicazioni come quelle bancarie o che hanno a che fare con i numeri in genere, è fondamentale tenere bene a mente che la culture fa la differenza.

Lavorando con le date, i problemi permangono se non prestiamo attenzione alla culture corrente, come evidenziato nell'Esempio 16.3.

Esempio 16.3

```
void DateTimeDemo()
{
   string dateAsString = "13.01.2018";
   DateTime date = DateTime.Parse(dateAsString);
   Console.WriteLine(date.ToString("D"));
}
```

L'Esempio 16.3 effettua il parsing di una stringa contenente una data, in un oggetto di tipo `DateTime` e, in un secondo momento, lo mostra a schermo. Nonostante ci sia già potenzialmente un problema di culture sull'interfaccia grafica per via del fatto che le date potrebbero mostrare giorni e mesi invertiti, passando una culture tipo "en-US" questa funzione fallirà già durante la chiamata a `DateTime.Parse` poiché il numero "13" verrà riconosciuto come mese e, ovviamente, il calendario gregoriano non prevede tredici mesi.

Una possibile soluzione a entrambi i problemi risiede nell'utilizzo di un overload dei metodi `Parse` e `ToString` che accettano un oggetto di tipo `IFormatProvider` come `NumberFormatInfo`, `DateTimeFormatInfo` e `CultureInfo`: specificando la culture, l'elaborazione sarà sempre identica e produrrà un risultato predicibile. Riprendendo l'Esempio 16.2, vediamo una sua possibile soluzione nell'Esempio 16.4.

Esempio 16.4

```
void NumbersDemo()
{
   string numberAsString = "10,500";
   decimal number = decimal.Parse(numberAsString, new CultureInfo("it-IT")) + 1;
   Console.WriteLine(number.ToString("C", new CultureInfo("en-US")));
}
```

L'ultimo esempio, molto simile ma leggermente diverso per soluzione, riguarda l'uso consistente della culture per quanto riguarda l'ordinamento.

Esempio 16.5

```
void SortingDemo()
{
   string[] states = { "Washington", "Virginia" };
   Array.Sort(states);
   foreach (string state in states)
   {
      Console.WriteLine(state);
   }
}
```

Considerando come esempio la lingua italiana, l'ordinamento è semplice perché è naturale considerare che la lettera "V" venga prima della lettera "W", ma esistono lingue, come il finlandese, in cui non c'è distinzione tra le due lettere. Pertanto, impostando una culture come "fi-FI", l'output generato risulterà Washington e poi Virginia, perché considerando identica la priorità tra i primi due caratteri, viene valutata la priorità tra i secondi ed evidentemente la lettera "a" viene prima della lettera "i". Stessa cosa si può ritenere valida per altre lingue, come per esempio il bosniaco, in cui le lettere accentate vengono valutate al pari delle stesse lettere senza accento. Anche in questi scenari, la soluzione richiede la sola aggiunta della tipologia di comparativa nel metodo `Sort`, come è illustrato nell'Esempio 16.6.

Esempio 16.6

```
void SortingDemo()
{
   string[] states = { "Washington", "Virginia" };
   Array.Sort(states, StringComparer.Ordinal);
   foreach (string state in states)
   {
```

```
      Console.WriteLine(state);
   }
}
```

Tra le possibili soluzioni evidenziate per risolvere i problemi di ordinamento, di numeri e date, si vanno ad aggiungere anche altre due culture di cui non abbiamo parlato: la culture neutrale e la `InvariantCulture`. La culture neutrale è piuttosto intuitiva: riguarda infatti una culture che ha assegnato nel suo tag il solo valore specifico della lingua ma non quello dell'area geografica, come nel caso di "it", che rappresenta in generale l'italiano. L'utilizzo di una culture neutrale è importante quando si vogliono supportare determinate varianti di una lingua ma non tutte, così il sistema è in grado di fare fallback in automatico sulla lingua principale: supponendo di avere l'inglese come lingua, dovremmo prevedere di gestire all'incirca più di quaranta varianti regionali differenti, come quella americana ("en-US"), quella inglese ("en-UK") o australiana ("en-AU") ma, per la maggior parte delle applicazioni, questo non è pensabile. Pertanto, per tutte le altre varianti che non supportiamo, viene comodo gestire in modo generico la lingua inglese ("en"), cosicché, se venisse richiesta, per esempio, la variante "en-ZW", ci sarebbero comunque dei contenuti localizzati.

Ogni culture ha previsto un oggetto padre, il parent, tramite il quale possiamo risalire alla culture più generica: supponendo, per esempio, di avere la variante "it-IT", il suo padre sarà la culture "it", senza la specifica dell'area geografica e questo è vero anche nei casi in cui, come per il bosniaco, venga specificato anche il dialetto. Si può risalire la catena dei parent fino a raggiungere la `InvariantCulture`, una sorta di culture neutrale alla quale però non viene assegnato un vero e proprio tag IETF e la lingua di default è l'inglese. Il padre di una `InvariantCulture` è la `InvariantCulture` stessa.

Esempio 16.7

```
static void Main(string[] args)
{
   CultureInfo.CurrentCulture = new CultureInfo("it-IT");
   while (CultureInfo.CurrentCulture != CultureInfo.InvariantCulture)
   {
      WriteLine($"CurrentCulture: {CultureInfo.CurrentCulture}");
      WriteLine($"IsNeutral: {CultureInfo.CurrentCulture.IsNeutralCulture}");
      WriteLine($"Parent culture: {CultureInfo.CurrentCulture.Parent}");
      WriteLine("**********");
      CultureInfo.CurrentCulture = CultureInfo.CurrentCulture.Parent;
   }
}
```

Nell'Esempio 16.7 possiamo notare come, indipendentemente dalla culture di partenza si risalirà sempre alla `InvariantCulture` e l'output generato è illustrato nella Figura 16.2.

La `InvariantCulture` può avere senso in quei casi in cui è obbligatorio specificare un parametro indicante la culture ma non si è troppo interessati a gestire l'eventuale valore e, pertanto, tutti gli eventuali metodi, come per esempio il `ToString`, tratteranno l'output nel formato standard inglese. Finora abbiamo parlato di quelle che sono le varie culture e di come .NET Core è in grado di gestirle. Per poter lavorare con le differenti lingue, però, c'è bisogno anche di qualche metodo, come quello offerto dai file di risorse, che consenta il salvataggio delle varie traduzioni.

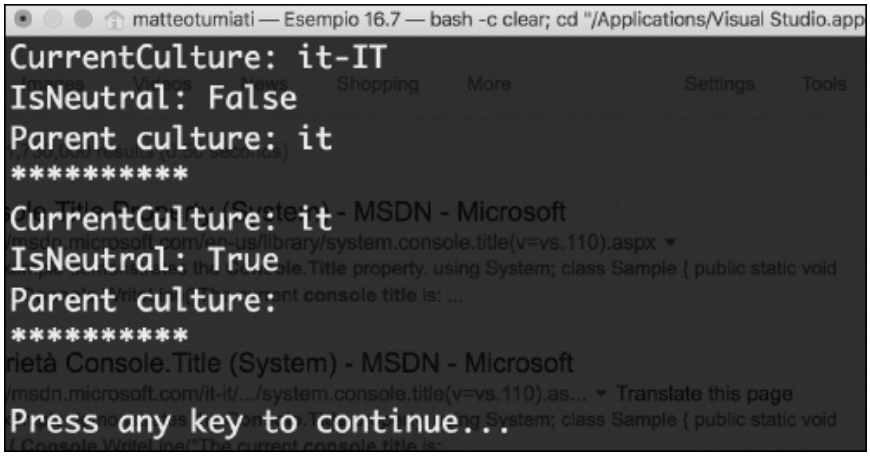


Figura 16.2 – La catena dei padri per il tag "it-IT" risale fino al raggiungimento della `InvariantCulture`.

Utilizzare i file di risorse

Un file di risorse è una sorta di file XML ed è probabilmente il sistema più semplice per salvare i contenuti che l'applicazione può utilizzare in un approccio multi-lingua: mentre link a file, immagini, icone e audio sono utilizzati in un ambiente legato al desktop, per il web ci si

concentra principalmente alla sola traduzione del testo. Per creare un nuovo file di risorse, è sufficiente cliccare con il tasto destro sul nome del progetto e aggiungere un nuovo file di tipo “Resource file”, come è illustrato nella Figura 16.3.

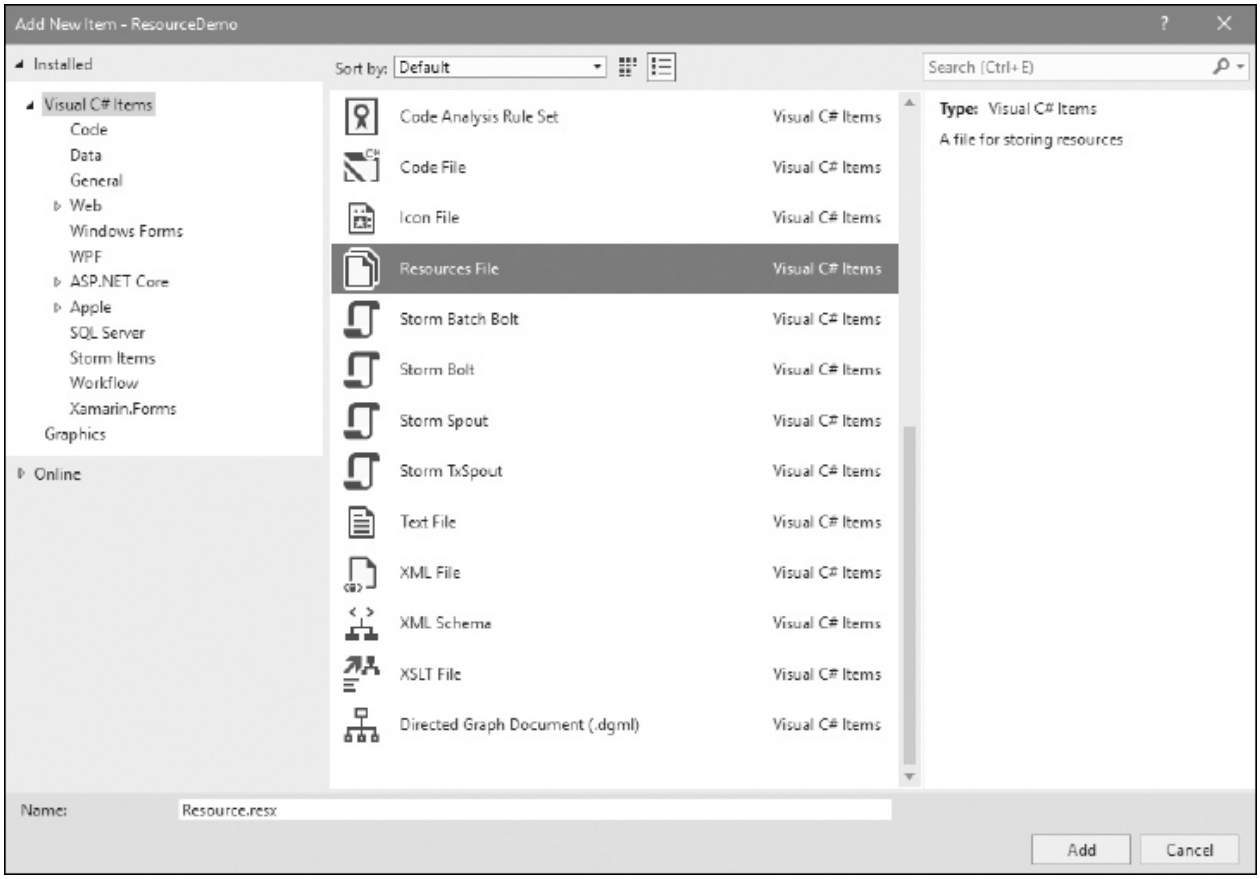


Figura 16.3 – Cliccando con il tasto destro sul progetto è possibile aggiungere un file di risorse.

Il file creato, come abbiamo già anticipato nel paragrafo precedente, è un file XML con una struttura predefinita e, all’interno di Visual Studio, c’è la possibilità di sfruttare il Managed Resources Editor per poterlo modificare senza dover mettere mano all’XML stesso, come viene mostrato nella Figura 16.4.

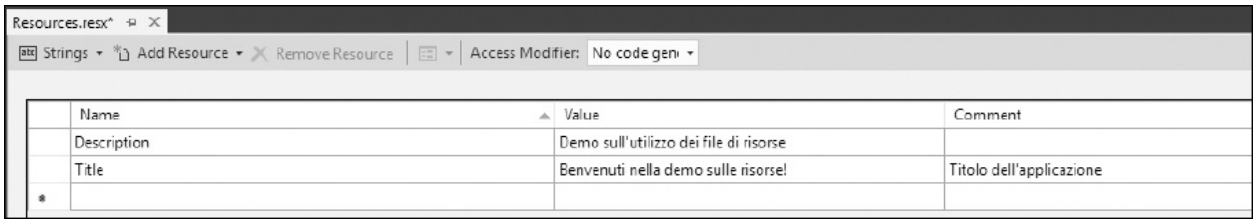


Figura 16.4 – I file di risorse possono essere modificati tramite l’editor visuale di Visual Studio (su Windows).

Il file generato contiene una serie di proprietà come `Name`, ovvero il nome della chiave che verrà utilizzato per referenziare tutte le varie traduzioni, `Value`, che rappresenta la vera e propria traduzione in una determinata lingua e `Comment`, che rappresenta un commento per descrivere meglio quella chiave, ma questo è un valore che può essere omesso. Il file `Resources.resx`, generato da Visual Studio, è composto da uno schema, da un paio di header e, infine, dalle vere e proprie traduzioni, che sono illustrate nell’Esempio 16.8.

Esempio 16.8

```
<data name=”Title” xml:space=”preserve”>
   <value>Benvenuti nella demo sulle risorse!</value>
   <comment>Titolo dell’applicazione</comment>
</data>
```

Saper modificare il valore direttamente tramite l’XML è importante poiché, come abbiamo già annunciato nei primi capitoli, ASP.NET Core e .NET Core in genere sono cross-platform, e pertanto quello che può essere sviluppato su Windows può essere continuato sull’ambiente Visual Studio for Mac di macOS, all’interno del quale però non è presente l’editor visuale e pertanto sarà necessario modificare il file di

risorse come se fosse un normale file XML. Tutte le modifiche che vengono fatte sul file di risorse, in modo indipendente dall'utilizzo dell'editor o dalla modalità manuale, vengono riflesse in automatico da Visual Studio su tutto il progetto perché c'è un watcher che rimane in ascolto di tutti i cambiamenti sul file e, tramite l'utilizzo di un altro file, il `Resources.Designer.cs`, auto-generato e chiamato file di code-behind, possiamo iniziare a referenziare tutte le chiavi, poiché vengono esposte come proprietà pubbliche e statiche all'interno del progetto, come viene mostrato nell'Esempio 16.9.

Esempio 16.9

```
static void Main(string[] args)
{
   Console.WriteLine(Resources.Title);
}
```

Il funzionamento del file di risorse non è cambiato rispetto al passato, dunque uno scenario di migrazione da un sistema esistente è sicuramente più semplice e veloce rispetto a un nuovo utilizzo. La lettura e, come vedremo in seguito, anche la scrittura di un file di risorse, è possibile tramite una classe chiamata `ResourceManager`, che si occupa di astrarre i contenuti del file di risorse e di gestire gli scenari relativi alle culture scelte, consentendo anche di fare il fallback in caso fosse necessario. Nel caso che abbiamo appena affrontato, però, non abbiamo parlato di lingue in modo specifico ma, anzi, abbiamo creato un file di risorse molto generico, come se fosse una sorta di dizionario chiave-valore sempre accessibile. In realtà, questo non è del tutto vero: abbiamo infatti creato un file di risorse che funziona per la `InvariantCulture` perché non è stata specificata alcuna culture. Per specificare che il file di risorse può essere utilizzato con una determinata lingua e area geografica, bisogna applicare una convenzione nel nome, ovvero bisogna creare un file `{nome-a-scelta}.{IETF-tag}.resx`, dove "{nome-a-scelta}" abbiamo visto essere `Resources` nel caso precedente, ma può essere cambiato con qualsiasi altro valore, e `{IETF-tag}` è il valore, opzionale, del language tag da utilizzare. Per creare un file di risorse per l'italiano, per esempio, si può creare un file `Resources.it-IT.resx` e, in base alla culture specificata nella `CurrentUICulture`, il `ResourceManager` decidere se utilizzare un file di risorse specifico per l'italiano piuttosto che quello per la `InvariantCulture`. Il codebehind per due file di risorse che hanno lo stesso "{nome-a-scelta}" non viene ricreato, poiché si presume che ci sia solo una variazione delle traduzioni e non delle chiavi applicate al file, pertanto rigenerarlo è considerato inutile.

È possibile riscrivere l'Esempio 16.9 facendo uso del `ResourceManager`, come è dimostrato nell'Esempio 16.10, in cui è possibile passare la `CurrentUICulture` come secondo parametro al metodo `GetString`, in modo da referenziare sempre la lingua corretta: i risultati ottenuti saranno identici, ma noteremo in modo più esplicito che cambiando la culture cambierà anche il risultato ottenuto e, qualora non ci sia la chiave ricercata, allora verrà richiamato il fallback.

Esempio 16.10

```
static void Main(string[] args)
{
   ResourceManager rm = new ResourceManager(typeof(Resources));
   var title = rm.GetString("Title", CultureInfo.CurrentUICulture);
}
```

Se andiamo a vedere il contenuto generato dal progetto in fase di build all'interno della cartella `bin`, noteremo che ci sono diverse cartelle con il nome del tag IETF, una per ogni lingua del file di risorse che abbiamo creato, al cui interno ci sarà una dll chiamata `{nome-del-progetto}.resources.dll`. Questa dll è chiamata assembly satellite perché ha solo il contenuto dei file di risorse auto-generato, non contiene codice, non può essere eseguita, può essere elaborata solo durante la compilazione, o a runtime dal `ResourceManager` e può essere iniettata in qualsiasi momento nell'applicazione. Al contrario di quanto abbiamo visto in Visual Studio, in cui ogni file di risorse specificava anche la lingua, l'organizzazione di questi assembly satellite è in cartelle, ma il risultato non cambia perché l'elaborazione è sempre trasparente grazie al `ResourceManager` e, in particolare, ad altre due classi utilizzate dal `ResourceManager` chiamate `ResourceReader` e `ResourceWriter`.

Poiché l'assembly generato dal file di risorse è una normale dll, non è possibile per le classi `ResourceReader` e `ResourceWriter` leggere o scrivere in modo diretto il file di risorse ma, al contrario, fanno uso di uno stream per caricare eventuali assembly pre-esistenti a runtime. Dobbiamo quindi continuare a utilizzare la `LoadFromAssemblyPath`, che prende in ingresso il percorso assoluto della dll e ritorna l'assembly come oggetto in memoria.

Esempio 16.11

```
void WriteResourceFile(string path)
{
   using (FileStream fs = File.OpenWrite(path))
   {
      using (ResourceWriter rw = new ResourceWriter(fs))
      {
         rw.AddResource("Title", "Titolo della risorsa");
         rw.AddResource("Description", "Descrizione della risorsa");
         rw.Generate();
      }
   }
```

```
}

void ReadResourceFile(string path)
{
   using (FileStream fs = File.OpenRead(path))
   {
      using (ResourceReader rr = new ResourceReader(fs))
      {
         IDictionaryEnumerator enumerator = rr.GetEnumerator();
         while (enumerator.MoveNext())
            Console.WriteLine($"{enumerator.Key} : {enumerator.Value}");
      }
   }
}
```

Nell'Esempio 16.11 si è dimostrato come sia possibile scrivere un file di risorse richiamando i metodi `AddResource` e `Generate` oppure leggere il file di risorse sfruttando un classico dizionario chiave-valore, in modo da scorrere tutte le traduzioni inserite. Una volta visto com'è possibile recuperare e lavorare tramite i file di risorse, è ora di iniziare a capire come ASP.NET Core è in grado di recuperare in automatico i contenuti da localizzare tramite l'uso di un'astrazione del `ResourceManager`.

La localizzazione dei contenuti

L'utilizzo dei file di risorse, nonostante sia il metodo più diffuso per quanto riguarda le applicazioni .NET, non è detto che sia il più efficiente o funzionale per quanto concerne le applicazioni moderne e che non devono essere migrate: per le nuove applicazioni, in particolare quelle sulla quale non c'è ancora ben chiara una idea di business e di espansione, può essere dispendioso iniziare a progettare un'applicazione subito in ottica di multi-lingua, pertanto c'è bisogno di metodi alternativi e che siano modificabili in qualsiasi momento. Proprio per questo motivo, e grazie anche al supporto della dependency injection nativa in ASP.NET Core, nascono i localizer, ovvero dei servizi che forniscono la localizzazione a un livello più alto rispetto al `ResourceManager`, che hanno già una implementazione di base nel framework ma sono anche personalizzabili secondo le proprie esigenze e permettono di lavorare non solo con i file di risorse ma anche con database SQL, piuttosto che dati esposti tramite file JSON oppure, oppure proprio per le applicazioni nuove, anche con dati in memoria. L'interfaccia `IStringLocalizer` espone una serie di indexer, proprietà e metodi che possono essere utilizzati dalle classi che la implementano (vedi `StringLocalizer`), in modo tale che si possa fare a meno di referenziare in maniera diretta il `ResourceManager`.

Esempio 16.12

```
LocalizedString this[string name] { get; }
LocalizedString this[string name, params object[] arguments] { get; }
public string Name { get; }
public string Value { get; }
public bool ResourceNotFound { get; }
```

L'Esempio 16.12 mostra alcune delle proprietà esposte dall'interfaccia `IStringLocalizer` che sono piuttosto intuitive: grazie a `Name` possiamo recuperare la chiave, con `Value` il valore della traduzione associata alla chiave, mentre `ResourceNotFound` indica se la risorsa è stata trovata in modo diretto oppure tramite il fallback, così che si possano prendere eventuali decisioni in merito.

Esempio 16.13

```
public interface ICustomService
{
   string SendResponse();
}

public class CustomService : ICustomService
{
   IStringLocalizer _localizer;
   public CustomService(IStringLocalizer<CustomService> localizer)
   {
      this._localizer = localizer;
   }
   public string SendResponse()
   {
      return _localizer["Title"];
   }
}
```

L'esempio 16.13 mostra come fare uso dell'interfaccia `IStringLocalizer` e come possiamo recuperare, tramite uno degli indexer, la traduzione corrispondente alla chiave `Title`. Nel caso questa non venga trovata, verrà restituita la chiave stessa, grazie all'implementazione di base fornita da `StringLocalizer`. Una particolarità che possiamo notare è che all'interno del costruttore non è stata iniettata `IStringLocalizer` ma una versione che fa uso dei generics: questa nuova versione eredita semplicemente dalla versione "normale" ma permette di essere utilizzata in uno scenario a dependency injection, per fare in modo che la stessa interfaccia non venga riutilizzata più volte. Quello che manca per completare è istruire il motore di dependency injection, per dirgli di caricare sia il servizio sia l'implementazione di `IStringLocalizer`, come viene mostrato nell'Esempio 16.14.

Esempio 16.14

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddScoped<ICustomService, CustomService>();
   services.AddLocalization();
}

public void Configure(IApplicationBuilder app,
         IHostingEnvironment env, ILoggerFactory loggerFactory)
{
   app.Run(async (context) =>
   {
      ICustomService service = context.RequestServices.GetService<ICustomService>();
      await context.Response.WriteAsync(service.SendResponse());
   });
}
```

Possiamo notare come non ci sia ancora un riferimento esplicito alla parte MVC di ASP.NET Core, e questo è normale perché non siamo obbligati a utilizzarlo. Inoltre, mentre è evidente che il `CustomService` è stato aggiunto alla lista dei servizi, sembra mancare completamente la registrazione per `IStringLocalizer`. La chiamata a `AddLocalization` si occupa di gestire, tra le altre cose, anche della registrazione, pertanto è più che sufficiente per avere una prima risposta. Nel caso in cui volessimo però riutilizzare i file di risorse che abbiamo visto prima, non c'è niente da fare se non prestare attenzione a un paio di dettagli:

- La convenzione dei nomi: ogni file di risorse può essere specifico per una singola classe e deve essere chiamato come `{namespace}.{nome-classe-generica<T>}.{tag-IETF}.resx` se vogliamo identificare la risorsa interamente nel nome, altrimenti ogni pezzo del namespace può diventare una cartella annidata fino ad arrivare all'ultimo livello dove viene specificato il solo tag IETF.

- Il percorso: di default ASP.NET Core si aspetta che le risorse appartengano alla root del progetto.

Poiché di solito i progetti diventano molto complessi con l'avanzare del tempo, per mantenere il progetto pulito e organizzato, possiamo specificare il percorso completo, a partire dalla root, dove saranno presenti i file di risorse tramite l'utilizzo delle `LocalizationOptions`, come viene spiegato nell'Esempio 16.15.

Esempio 16.15

```
services.AddLocalization(options =>
{
   options.ResourcesPath = "Resources";
});
```

Nonostante l'utilizzo sia piuttosto semplice, supponendo che il `CustomService` costruito sia un servizio che deve fare da dispatcher per altri servizi, diventa naturale immaginarsi tanti parametri di tipo `IStringLocalizer<T>` all'interno del costruttore, così da poterne specificare uno per servizio. Ma avere troppi parametri nel costruttore, specie se dello stesso tipo, non è una buona pratica. Per fortuna possiamo approfondire `IStringLocalizer` e vedere com'è fatta dietro le quinte: ogni volta che viene referenziata, il motore di IoC ritorna l'istanza reale, quindi `StringLocalizer`, che però ha nel costruttore un parametro di tipo `IStringLocalizerFactory` che, sfruttando nuovamente IoC, crea una classe di tipo `ResourceManagerStringLocalizerFactory`. Questa nuova classe è in grado di lavorare direttamente con il `ResourceManager` ed è da lì che arrivano, a tutti gli effetti, i contenuti tradotti per ogni chiave richiesta ed è qui che possiamo modificare il comportamento specificando, per esempio, di caricare le risorse da un file JSON piuttosto che da un database SQL. Include inoltre un metodo chiamato `Create`, che permette di creare al volo oggetti di tipo `IStringLocalizer`, secondo quella che è la classe di riferimento, come è dimostrato nell'Esempio 16.16.

Esempio 16.16

```
public interface ICustomService
{
```

```
   string SendResponse();
}

public class CustomService : ICustomService
{
   IStringLocalizerFactory _localizerFactory;
   public CustomService(IStringLocalizerFactory localizerFactory)
   {
      this.localizerFactory = localizerFactory;
   }
   public string SendResponse()
   {
      var localizer = localizerFactory.Create(typeof(AnotherService));
      return localizer["Title"];
   }
}
```

Nell'Esempio 16.16 si può notare come a ogni chiamata alla funzione `SendResponse` venga creata una nuova istanza, tramite il metodo `Create`, della factory stessa, per poi ritornare il valore corrispondente alla chiave `Title` inserito nel dizionario.

Nonostante nei paragrafi precedenti abbiamo già introdotto concetti come i file di risorse, utili per salvare le traduzioni nelle varie lingue, e le funzionalità di localizzazione tramite i vari `StringLocalizer`, ci manca ancora di capire come mostrare agli utenti che effettuano le richieste sul nostro sito web i contenuti localizzati nella lingua e culture richiesta.

Il localization middleware

Nei capitoli precedenti abbiamo già parlato nel dettaglio di quelli che sono i middleware e di come funzionano ma, oltre ai middleware più "standard" come quelli relativi a file statici, al routing o all'autenticazione, esiste anche un middleware dedicato alla localizzazione. Ogni chiamata HTTP viene processata da un thread diverso e il middleware della localizzazione lavora in thread isolation. Pertanto, ogni volta che vogliamo processare un determinato URL, siamo certi che verrà restituita la risposta corretta con la culture richiesta dall'utente, anche se le richieste in contemporanea fossero derivanti da più utenti e con più culture. Poiché deve essere garantita la thread isolation, scopriamo anche che nel motore di IoC i servizi `IStringLocalizer` e `IStringLocalizerFactory` sono stati registrati come transient. In questo modo, a ogni request viene generata – e iniettata nelle dipendenze che la richiedono – una nuova istanza.

Nonostante abbiamo già a disposizione dei file di risorse localizzati in più lingue e che abbiamo già visto come cambiare la lingua tramite l'impostazione delle proprietà `CurrentCulture` e `CurrentUICulture`, quello che manca è fare in modo che ASP.NET Core prelevi in automatico la lingua in base a quello che l'utente – o il sistema dell'utente – richiede. Un primo approccio potrebbe essere quello di sfruttare un parametro aggiuntivo nella querystring ma, per questioni di praticità e riusabilità, non è una scelta molto conveniente ed è per questo che entra in gioco il middleware `UseRequestLocalization`, come viene mostrato nell'Esempio 16.17.

Esempio 16.17

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   app.UseRequestLocalization();
   app.UseMvcWithDefaultRoute();
}
```

Una volta aggiunto il middleware alla pipeline di esecuzione, bisogna configurarne il comportamento. Pertanto, usiamo la classe `RequestLocalizationOptions` per impostare, tramite le sue proprietà pubbliche, quelle che sono le lingue che l'applicazione supporta e la lingua di default (in questo caso specifico, con lingua si intende sia la `CurrentCulture` sia la `CurrentUICulture`). L'Esempio 16.18 lo mette in evidenza.

Esempio 16.18

```
services.Configure<RequestLocalizationOptions>(options =>
{
   options.SupportedUICultures = new List<CultureInfo> {
      new CultureInfo("it"),
      new CultureInfo("en ")
   };
   options.FallBackToParentUICultures = true;
   options.DefaultRequestCulture = new RequestCulture("es");
});
```

Nel caso dell'Esempio 16.18, abbiamo impostato come culture supportate l'italiano e l'inglese ma, come culture di default, abbiamo impostato lo spagnolo: nel caso in cui la lingua richiesta sia l'italiano, non ci sono grossi problemi, in quanto è dichiarata come supportata

e verrà ricercata all'interno delle risorse disponibili, mentre, qualora facessimo una richiesta con una sua variante, per esempio l'italiano parlato in Svizzera ("it-CH"), il valore ritornato sarà ancora l'italiano, a meno che il fallback sia disabilitato e, in quel caso, verrebbero mostrati contenuti in lingua spagnola, se disponibili, altrimenti il valore previsto dal file di risorse generico per quel file, oppure, in ultima battuta, il valore della chiave di lookup.

Se ispezioniamo la richiesta fatta sul browser, ci accorgiamo che la culture richiesta viene recuperata tramite l'header `Accept-Language` che, in modo completamente automatico, viene valorizzato dal browser con tutti i valori derivanti dal sistema operativo e, con un ordine ben definito, tramite la variabile "q", che ne indica la qualità, viene scelto il valore che ha la preferenza (ovvero un valore di "q" prossimo a uno), come è illustrato nella Figura 16.5.

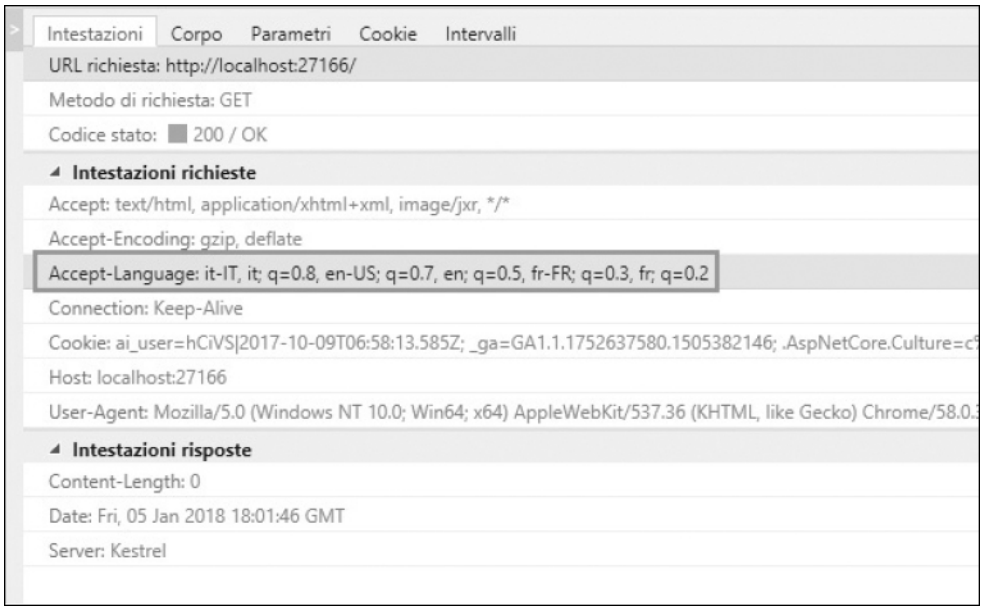


Figura 16.5 – L'header "`Accept-Language`" in una chiamata con il localization middleware abilitato mette in evidenza quali lingue possono essere utilizzare dal framework e secondo quale ordine d'importanza.

Se volessimo dare un po' di precedenza a quanto fatto dal browser, dovremmo intervenire in maniera diretta sulla querystring: passando il parametro `ui-culture`, oppure `culture` in modo indifferente, è possibile fare l'override di qualsiasi valore recuperato dall'header `Accept-Language` e l'intero processo di localizzazione continuerà con il valore della culture recuperato. Qualora il valore della querystring non esistesse o non fosse valido, verrebbe nuovamente recuperato il valore dall'header.

Entrambi i meccanismi discussi sono validi ma un po' estremi. Pertanto si potrebbe preferire una soluzione intermedia, in cui la culture venga specificata dall'utente e quindi salvata per le request successive: una soluzione basata a cookie potrebbe essere ideale e ASP.NET Core ha previsto un cookie di default, chiamato `.AspNetCore.Culture`, il cui valore è la concatenazione dei tag IETF per culture e culture dell'interfaccia, come è illustrato nella Figura 16.6.

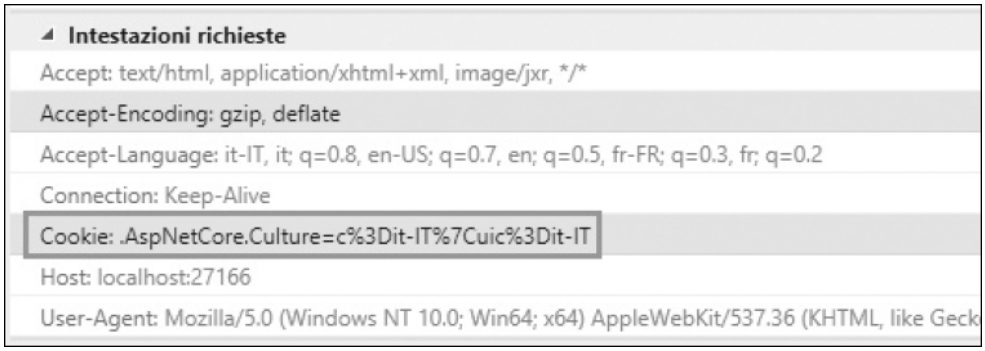


Figura 16.6 – Il cookie di default per la localizzazione di ASP.NET Core è utilizzato per salvare tutti i dettagli relativi alla culture e alla lingua corrente o richiesta durante la chiamata.

Per impostare il valore del cookie, è sufficiente crearlo con il nome predefinito da ASP.NET Core per la localizzazione (oppure uno personalizzato sempre basato sulla localizzazione) e restituirlo nella risposta, come è dimostrato nell'Esempio 16.19.

Esempio 16.19

```
RequestCulture requestCulture = new RequestCulture("it-IT");
var cookie = CookieRequestCultureProvider.MakeCookieValue(requestCulture);
Response.Cookies.Append(CookieRequestCultureProvider.DefaultCookieName, cookie);
```

Le tre pratiche che abbiamo visto sono tutte corrette ma la domanda da porsi ora è: qual è la strategia corretta da utilizzare per le nuove request? La risposta, come sempre nel mondo dell'IT, dipende dalla logica di business, quindi potrebbero essere tutte scelte valide e utilizzabili, così come potrebbero essere tutte sbagliate e potremmo voler preferire un sistema personalizzato. ASP.NET Core è un framework generico, pertanto non può conoscere le logiche di tutte le applicazioni e dà per scontato il fatto che tutte le tecniche che abbiamo visto siano valide e le implementa tramite Request Culture Provider: di default, verrà richiesto il `QueryStringRequestCultureProvider` che controllerà l'esistenza del parametro `ui-culture` o `culture` nella querystring. Qualora non dovesse trovarlo, farà uso del `CookieRequestCultureProvider` che controllerà se per caso esiste il cookie `.AspNet.Culture` e, se dovesse fallire anche quest'ultimo passaggio, proverà a invocare una istanza di `AcceptLanguageHeaderRequestCultureProvider` che farà caricare la culture tramite l'header `Accept-Language`. Per fortuna ASP.NET Core è un sistema formato da tante parti intercambiabili, e pertanto, se volessimo cambiare l'ordine dell'intero processo, sarà sufficiente modificare la proprietà `RequestCultureProviders` all'interno della configurazione delle `RequestConfigurationOptions`, come è dimostrato nell'Esempio 16.20.

Esempio 16.20

```
services.Configure<RequestLocalizationOptions>(options =>
{
   // Setup delle SupportedUICultures...
   // rimozione di tutti i provider di default
   options.RequestCultureProviders.Clear();
   // aggiunta dei provider nell'ordine personalizzato
   options.RequestCultureProviders.Insert(0, new
      AcceptLanguageHeaderRequestCultureProvider());
   options.RequestCultureProviders.Insert(1, new
      CookieRequestCultureProvider());
});
```

Nel caso in cui si vogliano creare URL con la localizzazione in modo che siano SEO-friendly, l'aggiunta del parametro `culture` nella querystring non è l'ideale ma, anzi, sarebbe meglio avere la culture già all'interno della route scelta. Come viene dimostrato nell'Esempio 16.21, per realizzare un sistema di questo tipo è sufficiente aggiungere il provider dedicato alla gestione delle route in prima posizione, così da dargli la giusta priorità, quindi bisogna creare una route dedicata a livello di MVC.

Esempio 16.21

```
public void ConfigureServices(IServiceCollection services)
{
// Aggiunta della localizzazione...

   services.Configure<RequestLocalizationOptions>(options =>
   {
      // Aggiunta delle SupportedUICultures...
      // Inserimento del provider dedicato
      options.RequestCultureProviders.Insert(0,
         new RouteDataRequestCultureProvider());
   });
   services.AddMvcCore();
}

public void Configure(IApplicationBuilder app)
{
   app.UseRequestLocalization();
   app.UseMvc(routes =>
   {
      routes.MapRoute(
         name: "defaultWithCulture",
         template: "{ui-culture}/{controller=Home}/{action=Index}/{id?}"
      );
   });
   app.UseMvcWithDefaultRoute();
}
```

Proseguendo su questa linea di principio, non è complesso realizzare un provider personalizzato: è solamente una classe che implementa l'interfaccia `IRequestCultureProvider`, ma poiché dipende dalla logica della propria applicazione, lasciamo al lettore l'implementazione concreta e la relativa documentazione ufficiale, disponibile all'indirizzo: http://aspit.co/bns.

Localizzare i contenuti e poterli gestire tramite i vari provider potrebbe già essere sufficiente per la maggior parte delle applicazioni; però esistono diversi scenari, come vedremo a breve, in cui le stesse view possono aver la necessità di mostrare componenti differenti.

La localizzazione delle view

Quello che abbiamo trattato finora si occupa in genere di localizzare i contenuti che possono essere serviti da un controller o, più in generale, da un servizio, ma quando si tratta di localizzare porzioni di view o, addirittura, codice HTML, bisogna introdurre un nuovo concetto: gli HTML localizer sono servizi che scrivono in modo diretto all'interno delle view e che non fanno uso dell'encoding in HTML di Razor. Il loro funzionamento, nonostante l'interfaccia `IHtmlLocalizer` non la ereditì in modo esplicito, è identico a quanto abbiamo visto già lavorando con `IStringLocalizer`, perché la creazione di un `IHtmlLocalizer` viene fatta tramite una istanza di `IHtmlLocalizerFactory` che, nell'implementazione predefinita, crea a tutti gli effetti uno oggetto di tipo `IStringLocalizer`. Poiché, però, viene richiesto in modo esplicito un qualcosa che possa lavorare con l'HTML, l'istanza creata viene wrappata all'interno di un oggetto che abbia una minima conoscenza dei meccanismi di rendering sulle view e che sia in grado di recuperare il codice HTML salvato nei file di risorse. Oltre all'uso, anche la registrazione è molto simile, ma richiede una variazione durante la fase di startup, come è mostrato nell'Esempio 16.22.

Esempio 16.22

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc().AddViewLocalization();
}
```

In questo caso si va a lavorare in modo esplicito con le view, pertanto la dipendenza da MVC è obbligatoria. All'interno della vista che ha i contenuti localizzati, basta iniettare la dipendenza su `IHtmlLocalizer`, specificando come tipo generico il controller di riferimento, come viene mostrato nell'Esempio 16.23. La risoluzione viene fatta tramite IoC in modo corretto poiché tutte le classi sono già state registrate con la chiamata all'extension method `AddViewLocalization` dell'esempio precedente.

Esempio 16.23

```
@model string
@inject IHtmlLocalizer<HomeController> localizer

<h1>@localizer["Title"]</h1>
<p>@localizer["Description", Model]</p>
```

Così come per la `ILocalizedString`, l'interfaccia `ILocalizedHtmlString` espone un indexer che accetta un secondo parametro opzionale a cui passare eventuali dati che vengono sostituiti ai placeholder contenuti nel valore associato alla chiave. Al contrario del valore stesso, però, i parametri opzionali non subiscono l'encoding. Per dimostrarlo, supponiamo di avere a disposizione una chiave `Title` il cui valore è "Benvenuto {0}": al posto del placeholder "{0}" verrà aggiunto, grazie alla view, il nome della persona che sta effettuando il login attraverso una form. Per capire se l'applicazione è vulnerabile a possibili attacchi di scripting, il primo test è provare a inserire al posto del nome la classica chiamata al metodo `Alert` di JavaScript, così, qualora dopo la `POST` della form si dovesse vedere il popup comparire, sapremmo che abbiamo dei problemi da risolvere. In realtà, come si può notare nella Figura 16.7, il contenuto non è stato elaborato ma solamente renderizzato come un normale campo di testo.

Figura 16.7 – Il rendering dei parametri associati al valore di una chiave di lookup con codice JavaScript non produce alcuna dialog poiché viene mostrato come solo testo.

Se invece cambiassimo il valore della chiave `Title` da "Benvenuto {0}" allo stesso script JavaScript appena passato come parametro, noteremo tutto un altro risultato.

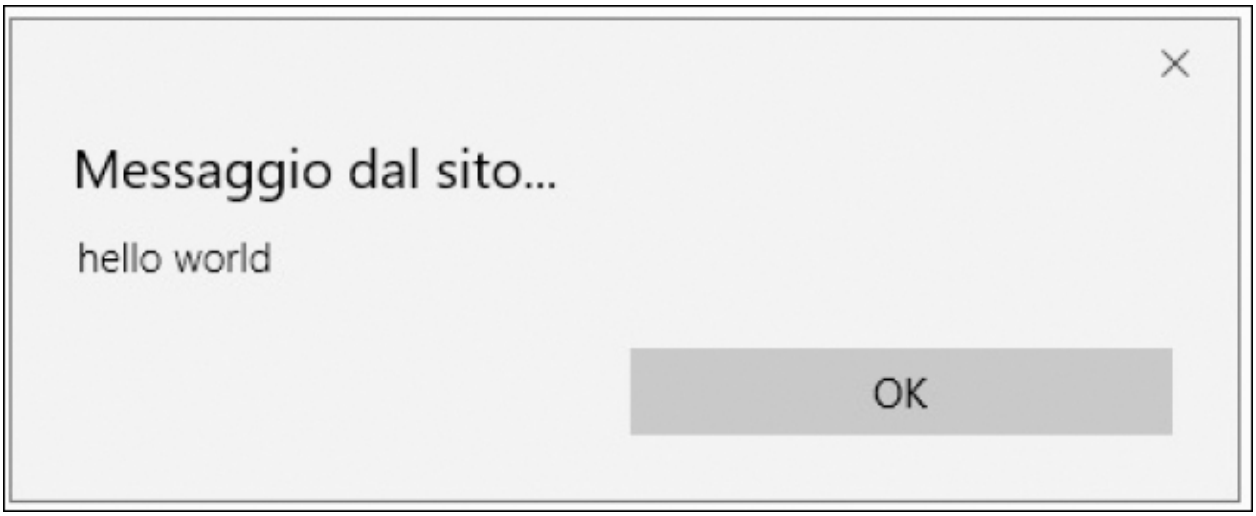


Figura 16.8 – Il rendering del valore di una chiave di lookup con codice JavaScript, invece, produce una dialog.

Questa differenza è dovuta al fatto che il render sta avvenendo molto presto nella costruzione della view, quindi tutto il JavaScript viene elaborato fin da subito, causando la comparsa del messaggio a schermo. Nonostante in questo caso non ci siano grossi problemi perché abbiamo scritto personalmente il file di risorse e quindi siamo noi stessi responsabili di eventuali problematiche a esso associate, a meno di applicazioni particolari, come i CMS, è fortemente consigliata la localizzazione di soli contenuti e non di codice HTML nelle risorse.

Un approccio molto simile a quello che abbiamo appena affrontato con gli HTML localizer lo si può avere con i view localizer, ovvero dei servizi che lavorano in modo specifico con Razor. L'interfaccia che verrà iniettata in questo caso è `IViewLocalizer`, che deriva in modo diretto da `IHtmlLocalizer` e non definisce nuove funzionalità e, pertanto, quanto visto finora è ancora valido. In più, però, eredita anche un'altra interfaccia, ovvero `IViewContextAware`, che ha a disposizione il metodo `Contextualize`, che viene utilizzato ogni volta che del contenuto deve essere mostrato nella view, così che, qualora ci fosse una dipendenza da qualche altro servizio durante l'utilizzo, il metodo `Contextualize` sarebbe in grado di risolverla a runtime, oppure, ancora, tramite la proprietà `ViewContext` potrebbe risalire a tutte le informazioni riguardanti i file di risorse associati. Ci sono un paio di differenze rispetto a quanto visto con gli HTML localizer:

- Area di funzionamento: c'è una forte dipendenza da Razor e `IViewLocalizer` eredita da `IViewContextAware`, pertanto se utilizzato in un controller, verrebbe lanciata una eccezione perché non sarebbe possibile per il controller accedere a informazioni relative alla view. Il servizio `IHtmlLocalizer`, invece, poiché viene utilizzato in modo esplicito tramite i generics con un controller associato, può essere iniettato anche nel controller stesso al posto di un `IStringLocalizer`.
- File di risorse: sia per `IStringLocalizer` sia per `IHtmlLocalizer` è necessario un file di risorse per controller, mentre per `IViewLocalizer` è necessario un file di risorse specifico per ogni view.

Per quanto riguarda il secondo punto, ovvero per la gestione dei file di risorse, non c'è una soluzione più valida di un'altra: infatti, creare un file unico con tutte le risorse piuttosto che un file specifico per ogni view è solamente una scelta di business, che non va a influire sul funzionamento ma solo sulla manutenibilità. L'ordinamento, ovvero come e dove devono essere caricate le view localizzate e quale nome devono avere per essere viste dal sistema di localizzazione, dipende da quanto viene impostato nella proprietà `LanguageViewLocationExpander`.

Esempio 16.24

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc()
      .AddViewLocalization(LanguageViewLocationExpanderFormat.Suffix);
}
```

Questa proprietà, come viene mostrato nell'Esempio 16.24, per default assume il valore `Suffix` e pertanto può essere omessa dal metodo `AddViewLocalization` e si aspetta che il nome della culture, ovvero il tag IETF, sia già integrato nel nome della view (per esempio "Index.it-IT.cshtml"). Tutto il resto della struttura non cambia, quindi verrà mantenuta in linea di massima una cartella per controller con tutte le view associate all'interno. In base a quanto visto nell'introduzione di questo capitolo, sarebbe opportuno prevedere in ogni caso dei meccanismi di fallback sia per le culture padri (per esempio "it" nel caso dell'italiano) sia per la `InvariantCulture` (il classico `Index.cshtml`). Se invece di voler creare intere view localizzate, volessimo creare solo dei file di risorse associati alle view, allora un file verrebbe ricercato in `{root-progetto}/Views/{nome-controller}/{nome-view}.{tag-IETF}.resx`.

Anche in questo caso vengono rispettati i meccanismi di fallback, quando la culture non venga recuperata al primo colpo e, inoltre, questi due sistemi possono lavorare assieme.

Nel caso in cui si preferisse una organizzazione in cartelle diversa da quella di default, bisognerebbe associare alla proprietà `LanguageViewLocationExpander` il valore `SubFolder`. L'unica differenza rispetto a quanto abbiamo appena detto per l'altro modello risulta nella definizione della struttura, che questa volta include i file in cartelle per tag IETF. Una vista "Index" potrebbe quindi essere ricercata in `Views/{nome-controller}/{tag-IETF}/Index.cshtml`, a meno che il percorso sia stato cambiato, come è illustrato nell'Esempio 16.15.

La localizzazione delle view va quasi a completare l'overview fatta, riguardante il supporto all'interno delle nostre applicazioni web, delle differenti culture. Nel caso in cui, per esempio, ci siano delle form di inserimento dati, però, potremmo trovare ancora qualche riferimento alla culture di default durante una verifica della validazione: per questo è necessario preoccuparsi anche della localizzazione delle data annotation.

Le data annotation

La validazione dei campi di una form, per esempio, viene fatta in due istanti temporali diversi: lato client, prima di effettuare il submit della form, e lato server, dove viene verificato che il modello sia valido. La validazione lato server è quella che affronteremo all'interno di questo capitolo poiché riguarda gli attributi di validazione, ovvero una parte delle data annotation, e di come questi attributi sfruttino il `ResourceManager` per recuperare le traduzioni dai file di risorse corrispondenti. La validazione lato client, invece, non sarà oggetto di questo capitolo poiché, nonostante sia ASP.NET a impostare gli attributi di validazione `data-*`, non viene gestita in modo diretto da ASP.NET Core e dipende principalmente dal sistema scelto, per esempio jQuery, che va a leggere ed elaborare quegli attributi.

Alcune delle data annotation più diffuse sono probabilmente quelle mostrate nell'Esempio 16.25, ovvero gli attributi `Required`, `MaxLength`, `Display` e `DataType`.

Esempio 16.25

```
[Required(ErrorMessage = "RequiredMessage")]
[MaxLength(50, ErrorMessage = "MaxLengthErrorMessage")]
[Display(Name = "FirstName")]
public string FirstName { get; set; }

[Required(ErrorMessage = "RequiredMessage")]
[DataType(DataType.EmailAddress, ErrorMessage = "EmailAddressErrorMessage")]
[Display(Name = "Email")]
public string Email { get; set; }
```

All'interno di tutte queste classi, c'è una proprietà che arriva dalla classe base `ValidationAttribute` chiamata `ErrorMessage`, che viene utilizzata e mostrata all'interno della form quando il modello inviato non è valido, cioè quando non rispetta gli attributi impostati. Questa proprietà è una stringa rappresentante un messaggio ma, qualora volessimo aggiungere la localizzazione, questo messaggio sarà la chiave utilizzata da ASP.NET Core per fare il lookup all'interno del file di risorse corrispondente al modello. La stessa cosa è valida per la proprietà `Name` relativa all'attributo `Display`, come viene mostrato nella Figura 16.9.

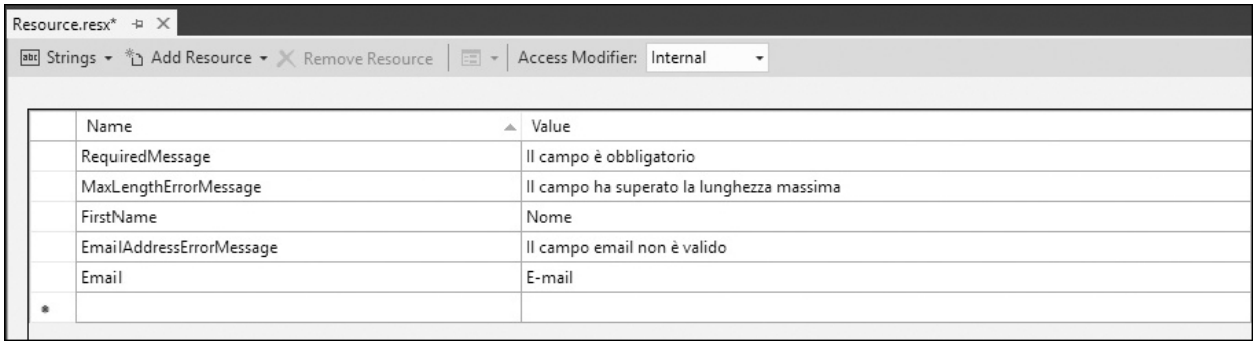


Figura 16.9 – Nel file di risorse per la localizzazione è anche possibile specificare i valori dei messaggi da assegnare alle data annotation.

Una volta creato il file di risorse associato al modello, la localizzazione continuerà a non funzionare poiché non è stato istruito il framework, quindi, come è mostrato nell'Esempio 16.26, registriamo la localizzazione per le data annotation nel file di startup.

Esempio 16.26

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc()
      .AddDataAnnotationsLocalization();
}
```

Siccome, come per le view, anche questa modifica si riflette tra modello e view stesse, c'è una forte dipendenza e bisogna registrare il servizio insieme a MVC.

L'organizzazione strutturale dei file di risorse non cambia rispetto a quanto abbiamo visto già per la localizzazione di contenuti e view ma, dal momento che in questo caso si rischiano di creare molti file – almeno uno per modello – e considerando che probabilmente si vuole visualizzare lo stesso messaggio di errore per tutti gli attributi dello stesso tipo (per esempio, per tutti gli attributi `Required` si potrebbe voler mostrare "Il campo è obbligatorio"), potrebbe venir comodo creare una classe intermediaria, utilizzata dal `ResourceManager`, così che le risorse vengano recuperate da un solo generico file (per tag IETF).

Ancora una volta, grazie alla modularità di ASP.NET Core e al suo funzionamento a plug-in, è possibile specificare il provider che si occuperà di localizzare le risorse, come viene mostrato nell'Esempio 16.27.

Esempio 16.27

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc()
      .AddDataAnnotationsLocalization(options =>
      {
         options.DataAnnotationLocalizerProvider = (type, factory) =>
         {
            return factory.Create(typeof(ErrorMessage));
         };
      });
}
```

Nell'esempio si può notare come il provider venga creato direttamente in linea, data la scarsa complessità; infatti, l'unica cosa che viene fatta è la creazione di una istanza tramite l'`IStringLocalizerFactory` di una classe `ErrorMessage` che è vuota ma che, come abbiamo anticipato, viene utilizzata dal `ResourceManager` per fare in modo che tutti i messaggi di errore (e in genere tutte le proprietà relative alle data annotation) vengano cercate nei file `ErrorMessage.{tag-IETF}.resx`.

Conclusioni

In questo capitolo abbiamo affrontato il tema dell'internazionalizzazione e abbiamo capito che è un approccio basato sulla globalizzazione, che avviene ancora prima dello sviluppo vero e proprio dell'applicazione, e sulla localizzazione, un processo che permette di rendere i contenuti adatti a una determinata lingua.

In termini di contenuti abbiamo visto quali sono le differenze tra applicare una localizzazione a un controller e a una view piuttosto che agli attributi di validazione dei modelli e si sono affrontate varie problematiche relative alle strategie di posizionamento dei file di risorse e alla loro modularità.

Utilizzando i middleware abbiamo costruito un'applicazione in grado di adattarsi ai vari agenti che effettuano le richieste, in modo che i contenuti serviti siano sempre coerenti in base al modello che è stato previsto, che sia tramite route piuttosto che tramite cookie oppure personalizzato.

La localizzazione è un aspetto importante delle applicazioni e non è da sottovalutare in un ambiente che ha bisogno di essere accessibile e fruibile da tutti. È un tema molto legato agli utenti finali che utilizzeranno l'applicazione e pertanto, come vedremo nel prossimo capitolo, è bene capire chi sono gli utenti che stanno utilizzando l'applicazione e quali ruoli hanno, per fornire loro contenuti sempre più personalizzati.

17
Autenticazione, autorizzazione e identità

Nel capitolo precedente abbiamo iniziato a vedere come personalizzare l'applicazione per offrire agli utenti una migliore esperienza di utilizzo, più vicina alle loro esigenze. Ora scopriremo come realizzare delle aree riservate protette da login per esporre funzionalità e contenuti in maniera selettiva, in base all'identità dell'utente. Realizzare un'applicazione del genere può sembrare semplice a prima vista ma presenta alcune criticità importanti, come conservare in maniera sicura i dati sensibili degli account locali e con la garanzia che non vengano inavvertitamente divulgati a estranei. Inoltre, quando un'applicazione deve inserirsi in un contesto aziendale, è probabile che debba integrarsi con un identity provider esterno, affinché gli utenti possano riutilizzare le loro credenziali e avere un'esperienza di Single Sign-On su tutte le applicazioni dell'organizzazione. Si parla quindi di identità federata, che è ormai nota al grande pubblico anche grazie al login con account social.

Un'applicazione moderna deve quindi essere in grado di supportare svariati meccanismi di autenticazione per consentire ai propri utenti di identificarsi con il minor sforzo possibile, per poi attuare precise policy di autorizzazione affinché le operazioni sui dati siano compiute soltanto da coloro che ne possiedono il privilegio. In questo capitolo vedremo come ASP.NET Core risponde a queste criticità e come ci permetta di costruire applicazioni sicure con il minimo sforzo.

Accedere a un'applicazione

Prima di addentrarci in dettagli tecnici, cerchiamo di comprendere in maniera approfondita cosa vuol dire accedere a un'applicazione. Si tratta di un'operazione che attraversa varie fasi ben distinte, che portano un utente anonimo a identificarsi mediante l'inserimento di credenziali, per poi essere riconosciuto e autorizzato a visitare i contenuti riservati dell'applicazione. La Figura 17.1 riepiloga ad alto livello queste fasi tipiche.

La Figura 17.1 riepiloga ad alto livello questo fasi tipiche:

- apertura di un'applicazione web e reindirizzamento alla pagina di login;
- inserimento di username e password e ottenimento del token di autenticazione;
- uso del token di autenticazione per l'accesso ai contenuti riservati dell'applicazione.

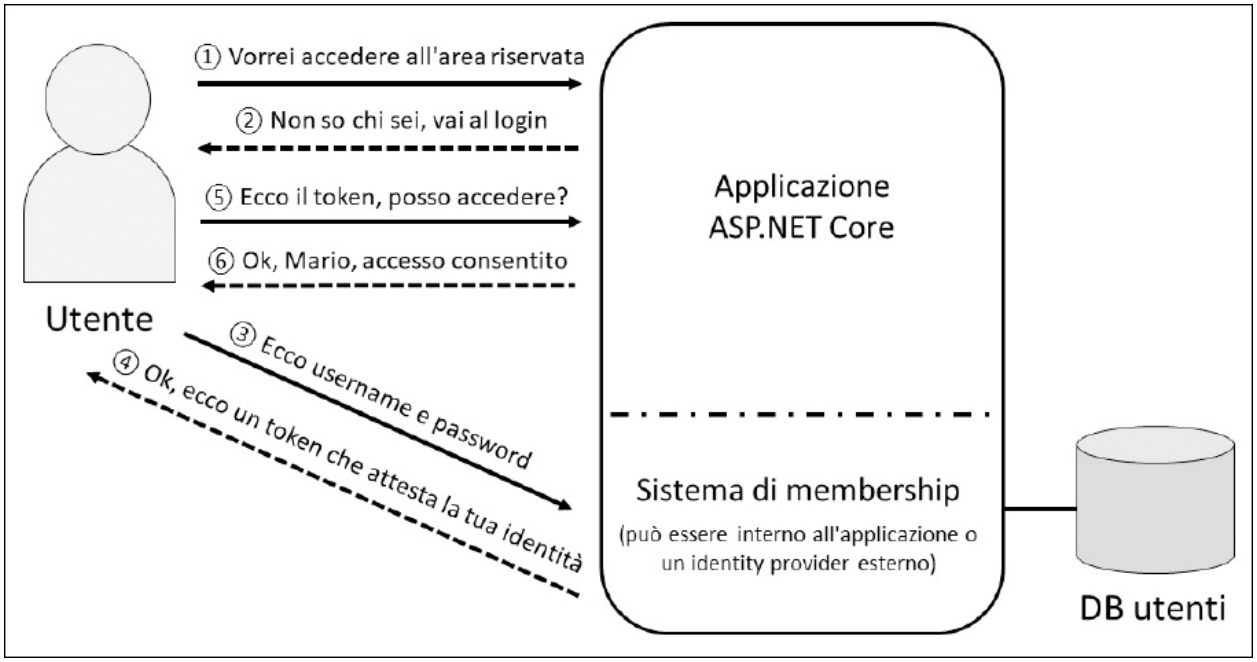


Figura 17.1 – Un utente anonimo accede all'applicazione solo dopo aver eseguito il login.

La fase di login

L'interazione di un utente inizia quando si presenta anonimamente all'applicazione e manifesta l'intenzione di voler entrare nella sua area riservata. Prima di poter accedere, però, l'utente è invitato a effettuare il login, ovvero identificarsi fornendo uno username e una password. L'effettiva modalità di conferimento di queste credenziali dipende dal tipo di applicazione che stiamo sviluppando:

- le web application presentano all'utente una pagina di login in HTML, in modo che egli possa digitare il suo username e la password nelle apposite caselle di testo;
- le web API espongono un token endpoint a cui il client invia le credenziali. Esempi di client della web API sono le app mobile o le Single-Page Application sviluppate in JavaScript.

Durante la fase di login, l'applicazione sfrutta un sistema di membership per verificare se lo username e la password corrispondono a un utente nel database locale. In alternativa, può delegare questo compito a un identity provider esterno. In entrambi i casi, se il login ha successo, viene emesso un cookie o un token JWT che attesta l'identità dell'utente e che il client dovrà presentare all'applicazione in ogni successiva richiesta, per identificarsi senza dover fornire di nuovo le credenziali.

Un'identità formata da claim

I cookie e i token JWT emessi all'atto del login contengono un determinato numero di claim, ovvero informazioni personali che descrivono l'identità dell'utente loggato. Ogni claim è una coppia chiave-valore che ne indica il tipo come il ruolo o il nome e il relativo valore assegnato all'utente corrente. L'identità, così serializzata nel cookie o nel token JWT, è cifrata o firmata digitalmente in modo che il client non possa manipolarla senza comprometterne l'integrità. Il tutto è corredato da una data di scadenza, proprio come un qualsiasi documento d'identità, illustrato nella Figura 17.2.

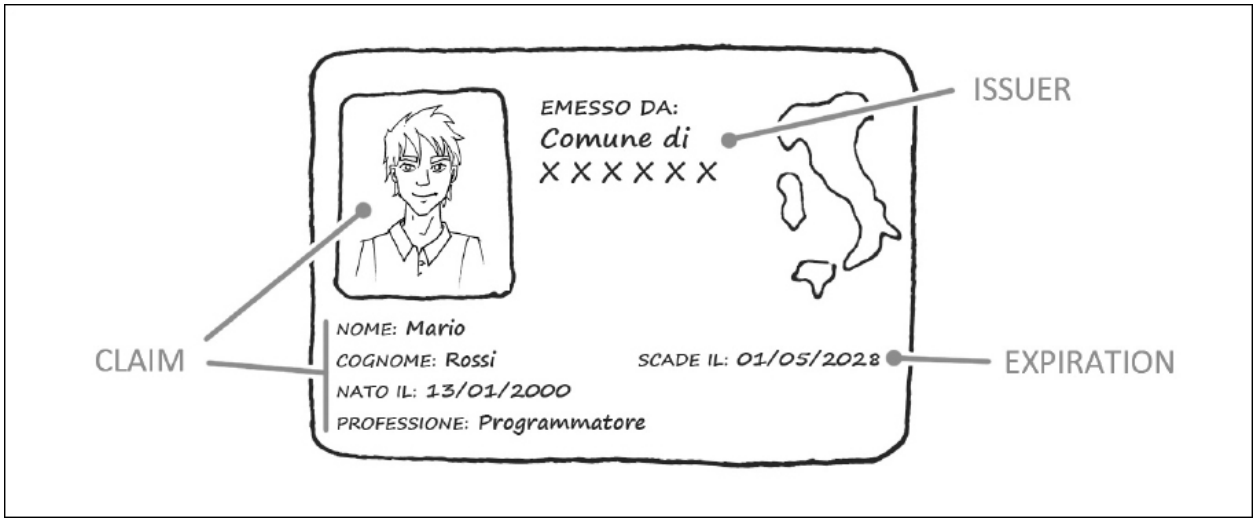


Figura 17.2 – Come un documento d'identità, i token e i cookie contengono dei claim e hanno una data di scadenza.

In questo modo, concediamo al client di usare un titolo di autenticazione limitato nel tempo, che sarà riemesso periodicamente con informazioni aggiornate. L'effettiva scadenza è configurabile in base allo specifico scenario di utilizzo: da pochi minuti fino ad alcuni giorni o mesi.

La fase di autenticazione

Quando il client presenta il cookie o il token JWT nella sua richiesta, vengono verificate integrità e scadenza prima che i claim possano essere estratti. In un'applicazione ASP.NET Core, questi compiti sono svolti dal middleware di autenticazione, che si occupa anche di usare i claim per costruire un oggetto di tipo `ClaimsIdentity` a rappresentazione dell'identità dell'utente. Il middleware supporta diversi meccanismi di autenticazione, che esamineremo nel corso del capitolo, e perciò è in grado di creare molteplici `ClaimsIdentity`, una per ogni titolo di autenticazione fornito dal client. Esse sono quindi incapsulate in un unico oggetto `ClaimsPrincipal` come si può vedere nella Figura 17.3.

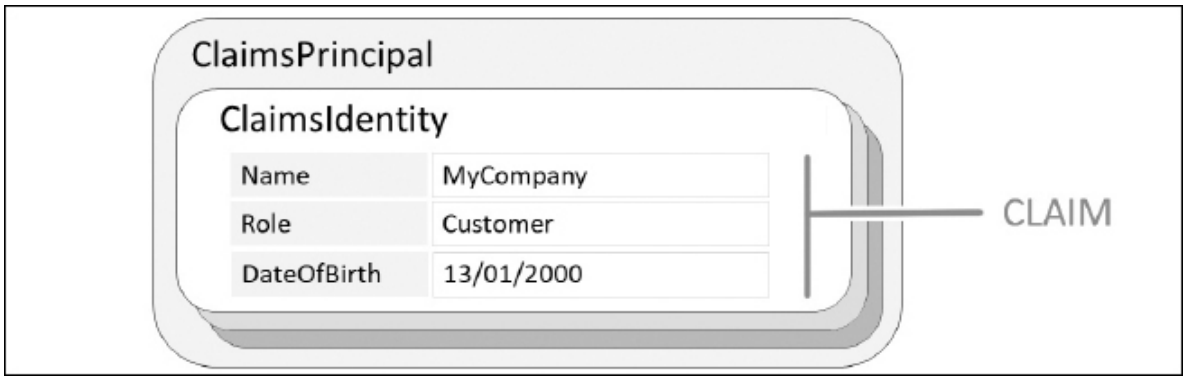


Figura 17.3 – L'oggetto ClaimsPrincipal incapsula una o più ClaimsIdentity, ciascuna contenente dei claim.

Così creato, il `ClaimsPrincipal` viene assegnato alla richiesta HTTP corrente e l'accompagnerà per tutto il resto della sua elaborazione. In questo modo, la richiesta è autenticata e tutti i successivi middleware della pipeline potranno compiere decisioni in base ai claim trovati nelle identità, come nella Figura 17.4.

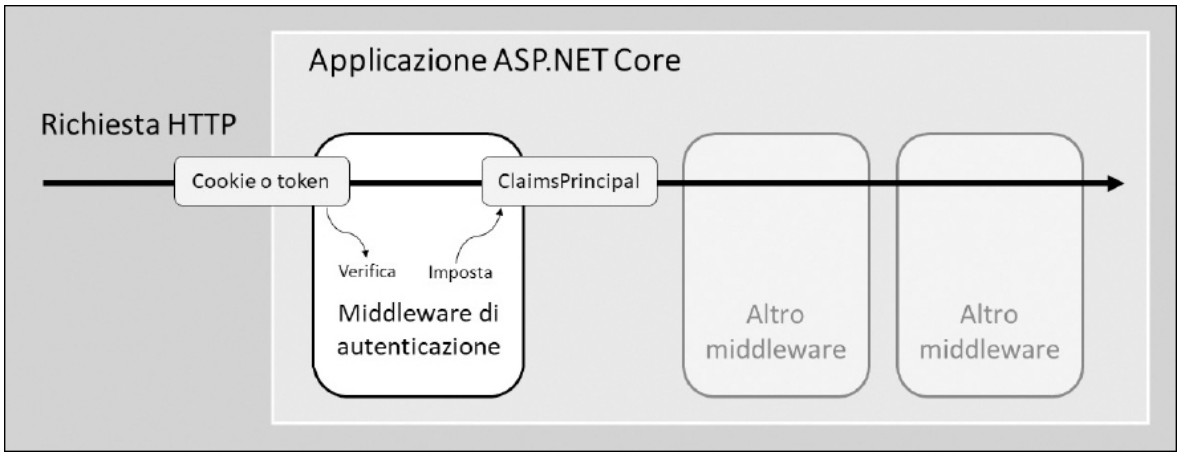


Figura 17.4 – Il middleware di autenticazione imposta un ClaimsPrincipal per la richiesta HTTP corrente.

Dato che il `ClaimsPrincipal` resta in memoria per tutta la durata della richiesta, ogni componente dell'applicazione può leggerla dalla proprietà `HttpContext.User`. Le informazioni contenute nei claim sono subito accessibili ed evitano all'applicazione di dover interrogare il database per i compiti più elementari, come visualizzare nel sito le informazioni dell'utente loggato.

La fase di autorizzazione

Creare un `ClaimsPrincipal` non basta: è necessario verificare se i claim contenuti in essa danno diritto all'utente di accedere alla risorsa richiesta. Questa fase di autorizzazione è importantissima e impedisce agli utenti di compiere operazioni per cui non possiedono il privilegio, come nella Figura 17.5.

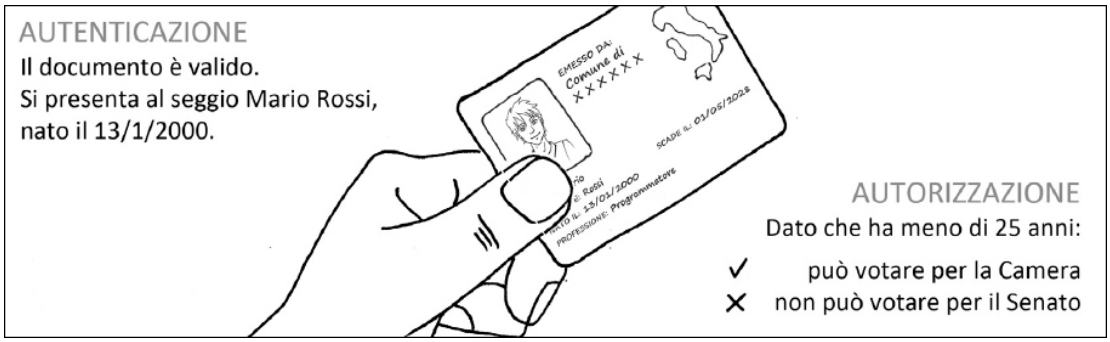


Figura 17.5 – L'utente è autorizzato (o no) a compiere operazioni in base ai suoi claim, come l'età.

Il fatto che l'utente si sia autenticato, infatti, non sempre è una condizione sufficiente a consentirgli un accesso a qualsiasi funzionalità offerta dall'applicazione. Tipicamente, le applicazioni web prevedono varie tipologie di utenti: un cliente, per esempio, può inviare un ordine ma solamente un "amministratore" può richiedere il riassortimento della merce ai fornitori. In un'applicazione ASP.NET Core, la fase di autorizzazione può essere svolta da un middleware, che ha la facoltà di bloccare la richiesta nel caso in cui non sussistano i claim necessari. La Figura 17.6 illustra questo concetto.

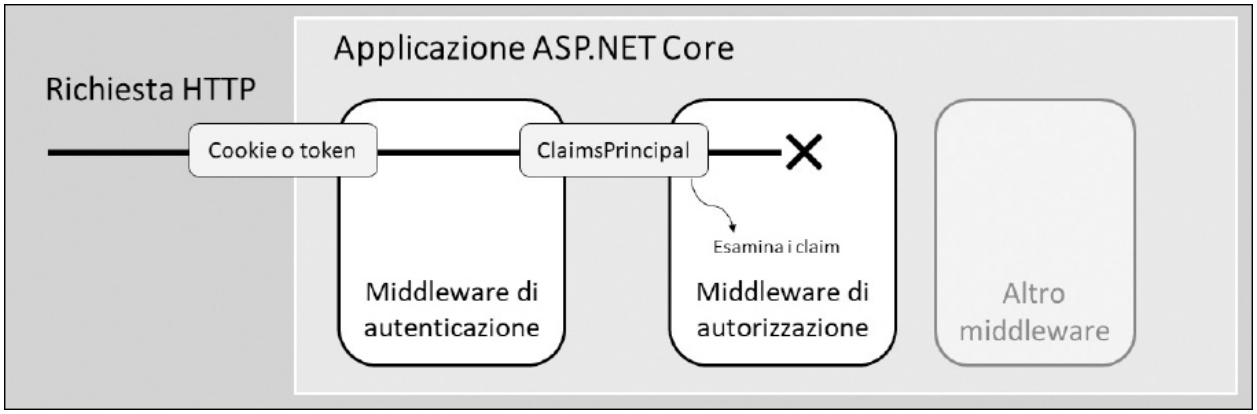


Figura 17.6 – Un middleware di autorizzazione può bloccare la richiesta se l'utente non ha i claim idonei.

Più tipicamente, in applicazioni ASP.NET Core MVC e Web API, l'autorizzazione viene configurata per mezzo dell'attributo `AuthorizeAttribute`, di cui parleremo ampiamente nel corso di questo capitolo.

Scegliere un sistema di membership

Durante la fase di login, l'applicazione deve essere in grado di verificare se le credenziali corrispondono a quelle di un utente presente nel database e, in caso affermativo, emettere il cookie o il token JWT di autenticazione. Con ASP.NET Core, siamo liberi di realizzare questa funzionalità come preferiamo ma, quasi sempre, questo porta con sé dei rischi. Password memorizzate in chiaro, nessuna protezione da tentativi d'intrusione e mancanza di una funzionalità di logout remoto sono solo alcuni dei difetti che potremmo introdurre nell'applicazione se decidiamo di creare un nostro sistema di membership personalizzato. Quello di cui abbiamo bisogno è di un sistema sicuro, costruito per essere conforme alla normativa europea del GDPR, così da garantire ai nostri utenti la massima protezione e trasparenza nel trattamento dei dati, caratteristiche imprescindibili in una applicazione web moderna. Vediamo dunque quali sono le migliori opportunità per gestire l'accesso con utenti locali.

Membership con ASP.NET Core Identity

Quando vogliamo avere il pieno controllo sul database su cui archiviare i nostri utenti, ASP.NET Core Identity è la scelta ideale perché è una soluzione estremamente estendibile e che perciò si può adattare a svariate esigenze. È un sistema di membership in-app completo e sicuro, che si avvale sia dell'esperienza di Microsoft sia dei contributi e delle segnalazioni dei membri della community. Le funzionalità che offre possono essere raggruppate in due insiemi principali:

- un servizio di persistenza degli utenti su database locale, comprensivo dei meccanismi di cifratura e verifica delle password. Gli utenti possono essere inseriti, aggiornati ed eliminati grazie alla API `UserManager<TUser>` che esamineremo in dettaglio nel capitolo 18;
- una UI personalizzabile che consente agli utenti di compiere tutte le operazioni legate al loro profilo, tra cui registrarsi, recuperare la password, effettuare il login e modificare i propri dati personali, con una conseguente riduzione dei tempi di sviluppo.

ASP.NET Core Identity è anche facile da integrare nell'applicazione, grazie agli esempi forniti dai template di .NET Core. Per iniziare, creiamo un nuovo progetto `mvc` o `webapi` con il comando illustrato nell'Esempio 17.1. Lo stesso progetto con gestione degli account individuali può ovviamente essere creato anche con il wizard grafico di Visual Studio e Visual Studio for Mac.

Esempio 17.1

```
dotnet new mvc --auth Individual
```

Il parametro `--auth Individual` ci assicura che ASP.NET Identity Core verrà incluso nel progetto, con archiviazione degli utenti su database SQLite. Se preferiamo usare SQL Server, aggiungiamo il parametro `--use-local-db` in coda al comando e poi personalizziamo la connection string che per default si trova nel file `appsettings.json`.

All'interno nel progetto, nel metodo `ConfigureServices` della classe `Startup`, troveremo le righe di codice per la configurazione di ASP.NET Core Identity, come si vede nell'Esempio 17.2.

Esempio 17.2

```
services.AddDefaultIdentity<IdentityUser>()
          .AddEntityFrameworkStores<ApplicationDbContext>();
```

L'extension method `AddDefaultIdentity` configura i servizi usati da ASP.NET Core Identity e richiede che sia fornito il tipo che rappresenta l'utente nella nostra applicazione. `IdentityUser` è una classe che implementa il comportamento minimo, ma può essere estesa o sostituita del tutto con implementazioni personalizzate, come vedremo nel corso del prossimo capitolo.

Per specificare lo storage provider usato per persistere gli utenti nel database locale, usiamo l'extension method `AddEntityFrameworkStores` e indichiamo il tipo del `DbContext` che si trova nel nostro progetto. In questo modo useremo Entity Framework Core per preparare il database degli utenti, che potrà essere creato a nostra scelta su Sql Server, Sqlite, MySql o uno degli altri provider supportati. Se non desideriamo usare Entity Framework, ASP.NET Core Identity può funzionare con ulteriori provider, come quello per MongoDB realizzato dalla community. Un elenco esaustivo dei provider disponibili è pubblicato all'indirizzo http://aspit.co/bpl.

A partire da ASP.NET Core 2.1, la UI di gestione dell'account viene veicolata come pacchetto NuGet, in modo che sia facilmente aggiornabile. Le sue pagine non appariranno dunque all'interno del codice sorgente del progetto ma potranno comunque essere personalizzate, come scopriremo nel prossimo capitolo. La Figura 17.7 mostra un esempio una delle pagine di gestione del profilo.

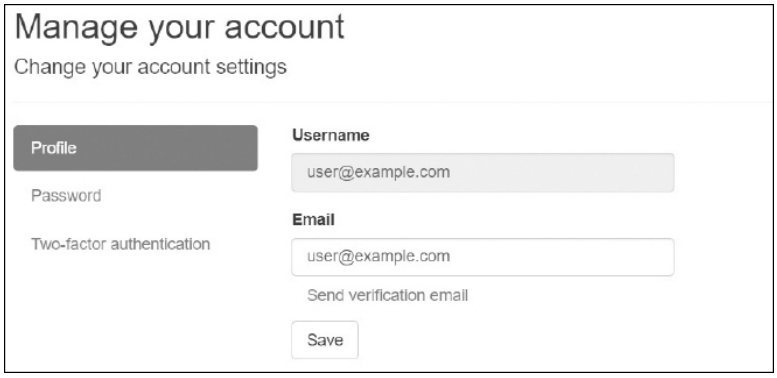


Figura 17.7 – La UI di ASP.NET Core Identity fornisce all'utente le pagine per gestire il suo account.

L'extension method `AddDefaultIdentity` si occupa anche di registrare i vari servizi usati da ASP.NET Core Identity, come quelli incaricati della generazione dei token di sicurezza inviati all'utente via e-mail o sms per la conferma dell'account o per il recupero della password. Questa configurazione predefinita è idonea per la maggior parte delle applicazioni ma in rari casi potremmo voler usare l'extension method `AddIdentityCore`, che invece fornisce solo la funzionalità di gestione degli account ed eventualmente usare la sua fluent interface per aggiungere altre funzionalità desiderate.

Membership con Azure Active Directory B2C

In alternativa ad ASP.NET Core Identity, la nostra applicazione può avvalersi di un identity provider esterno come Azure Active Directory B2C, un sistema di membership ospitato nel cloud. Delegare la gestione degli utenti a un servizio cloud significa sfruttare un'infrastruttura con sicurezza di livello enterprise, che può facilmente scalare per gestire milioni di account. Azure AD B2C offre una maggiore protezione contro il furto degli account grazie all'impiego di algoritmi di intelligenza artificiale, che sono in grado di identificare i tentativi di accesso non autorizzato e ridurli drasticamente. Proprio come ASP.NET Core Identity, anche Azure AD B2C espone un'interfaccia web che l'utente può usare per registrarsi, effettuare il login, modificare i dati del suo profilo o recuperare la password. L'aspetto grafico delle pagine è completamente personalizzabile, così che non siano percepite discontinuità di design al reindirizzamento verso la pagina di login nel cloud, illustrata nella Figura 17.8.

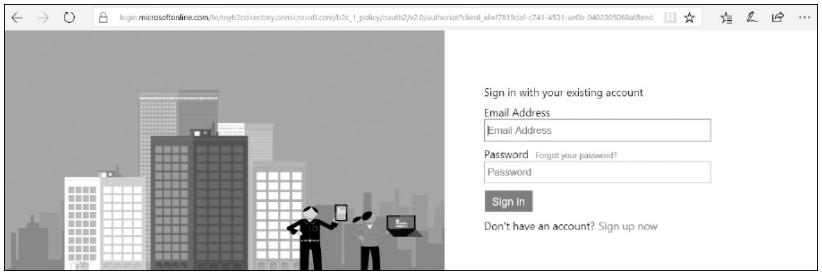


Figura 17.8 – Con Azure AD B2C, l'utente compie il login da una UI personalizzabile ospitata nel cloud.

Azure AD B2C permette l'accesso con account locale, ma può anche supportare identity provider esterni e i principali social network. Per questo motivo, è un servizio idoneo a un pubblico eterogeneo, formato sia da utenti consumer sia appartenenti a organizzazioni. Infine, il portale di Microsoft Azure offre svariati strumenti all'amministratore, che in questo modo può configurare i claim richiesti all'atto della registrazione, gestire i profili dei suoi utenti e tenere sotto controllo il log degli accessi. Le istruzioni per creare il proprio tenant di Azure Active Directory B2C sono riportate nella documentazione ufficiale, all'indirizzo: `http://aspit.co/bmz`.

Per creare una nuova applicazione ASP.NET Core che sfrutti Azure AD B2C come sistema di membership, digitiamo il comando riportato nell'Esempio 17.3.

Esempio 17.3

```
dotnet new mvc --auth IndividualB2C
```

All'interno della classe `Startup` troveremo già presente tutta la configurazione necessaria. L'unico intervento consisterà nel modificare i valori presenti nel file `appsettings.json`, sostituendoli con quelli relativi al nostro tenant, ottenuti dal portale di Microsoft Azure.

Configurare l'autenticazione: account locali

Se scegliamo di gestire gli account in un database locale grazie ad ASP.NET Core Identity o a un sistema di membership personalizzato, la nostra applicazione ha l'onere di emettere un titolo di autenticazione in seguito al corretto login dell'utente. Il modo in cui tale titolo viene veicolato al client cambia in base al tipo di applicazione che stiamo sviluppando:

- nelle web application, viene emesso un cookie di autenticazione in cui è serializzata l'identità dell'utente e che il browser client memorizzerà localmente, per poi restituirlo in tutte le successive richieste, automaticamente;
- anche nelle web api è tecnicamente possibile usare un cookie di autenticazione, in special modo quando il client è una Single-Page Application che viene eseguita nel browser dell'utente. Tuttavia, una web api può agire da backend anche per altri tipi di applicazioni, come le app mobile native o i dispositivi IoT, piattaforme sulle quali gestire i cookie può risultare leggermente più difficoltoso. In questi casi, un token JWT rappresenta un'alternativa migliore perché non deve essere interpretato ma consiste di una stringa che il client restituirà tale e quale, come parte delle intestazioni o del corpo della richiesta HTTP. Un token JWT, proprio come un cookie, contiene l'identità dell'utente e possiede una data di scadenza.

Il middleware di autenticazione di ASP.NET Core supporta vari meccanismi, anche definiti authentication scheme, da abilitare e configurare secondo le esigenze della nostra applicazione.

Prima di decidere quali scheme abilitare, aggiungiamo innanzitutto il middleware alla pipeline, usando l'extension method UseAuthentication dal metodo `Configure` della classe `Startup`, come è illustrato nell'Esempio 17.4. Se abbiamo creato il progetto da un template con gestione degli account individuali, questa configurazione sarà già presente.

Esempio 17.4

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   //Agiungiamo il middleware di autenticazione di ASP.NET Core
   app.UseAuthentication();
   //Qui configurazione di altri middleware
}
```

Ricordiamo che il middleware di autenticazione di ASP.NET Core si occupa di esaminare la richiesta HTTP per verificare se il client sta fornendo un titolo di autenticazione per mezzo di uno degli scheme abilitati. In caso positivo, il middleware ne verifica integrità e scadenza, per poi creare un `ClaimsPrincipal` che viene associato alla richiesta corrente.

Vediamo ora come configurare il middleware affinché supporti gli authentication scheme basati su cookie e su token JWT.

Autenticazione su cookie con ASP.NET Core Identity

Dato che l'autenticazione basata su cookie è di utilizzo comune nelle web application, la troviamo già abilitata in ogni nuovo progetto che usa ASP.NET Core Identity. Anche se non è richiesta alcuna modifica al codice da parte nostra, in determinate situazioni può essere necessario modificare le opzioni di emissione del cookie, come la sua scadenza o il dominio di validità. L'Esempio 17.5 mostra l'uso dell'extension method `ConfigureApplicationCookie` per modificare tali opzioni. Per esempio, impostando l'opzione `ExpireTimeSpan` possiamo ridurre la scadenza dei cookie a un solo giorno, ovvero una impostazione più restrittiva rispetto ai 14 giorni di default.

Esempio 17.5

```
public void ConfigureServices(IServiceCollection services)
{
   services.ConfigureApplicationCookie(options =>
   {
      //Sliding expiration di un giorno
```

```
        options.ExpireTimeSpan = TimeSpan.FromDays(1);
        //Abilitiamo questo se vogliamo impostare una absolute expiration
        //options.SlidingExpiration = false;
    });
    //Qui configurazione di altri servizi
}
```

L'applicazione ASP.NET Core emetterà automaticamente un nuovo cookie con scadenza aggiornata quando quello precedente ha superato più di metà della sua vita, così da garantire all'utente una continuità di utilizzo. Se invece intendiamo impostare una scadenza assoluta e non rinnovabile, possiamo valorizzare l'opzione `SlidingExpiration` su `false`. Alla scadenza del cookie, l'utente dovrà necessariamente rieseguire il login. Teniamo a mente che queste impostazioni avranno effetto solo selezionando la checkbox "Remember me" illustrata nella Figura 17.9, causando così l'emissione di un cookie persistente. In caso contrario, verrebbe emesso un cookie di sessione, ovvero un cookie che il browser eliminerà automaticamente alla sua chiusura.

Figura 17.9 – Le impostazioni sulla scadenza del cookie hanno effetto solo selezionando "Remember me".

Per verificare che l'applicazione stia correttamente emettendo cookie secondo le nostre disposizioni, possiamo usare gli strumenti di sviluppo del browser, come viene mostrato nella Figura 17.10.

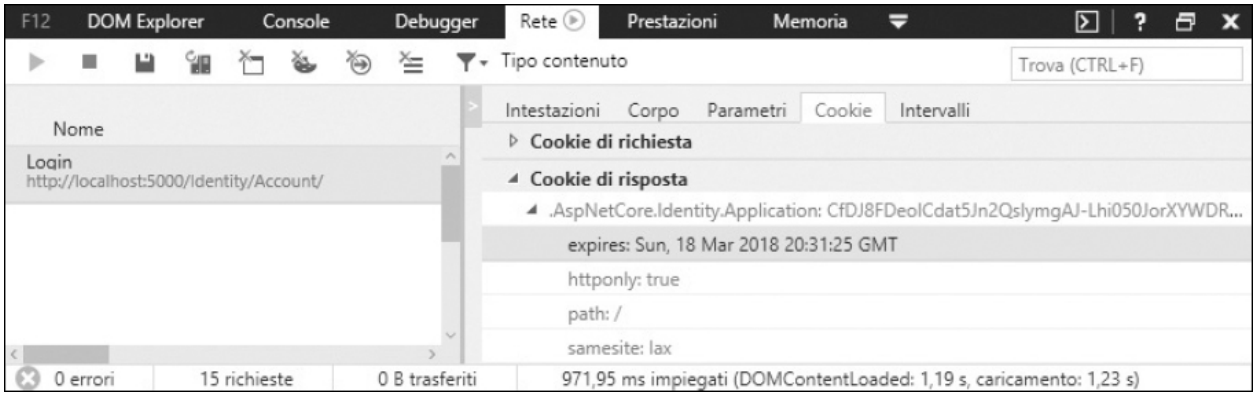


Figura 17.10 – Un cookie persistente si riconosce perché possiede una data di scadenza ben definita.

Il contenuto del cookie viene cifrato da ASP.NET Core in modo che sia protetto dalla contraffazione. Ogni tentativo di alterare il cookie da parte del client ne comprometterebbe l'integrità.

Autenticazione su cookie senza ASP.NET Core Identity

Nel caso in cui decidessimo di fare a meno di ASP.NET Core Identity, possiamo comunque sfruttare l'autenticazione basata sui cookie, così come qualsiasi altro scheme supportato da ASP.NET Core. Il middleware di autenticazione, infatti, può essere impiegato anche in applicazioni che usano sistemi di membership completamente personalizzati. Iniziamo dal metodo `ConfigureServices` della classe `Startup`, dove usiamo l'extension method `AddAuthentication` per indicare gli scheme che intendiamo supportare, grazie alla comoda fluent interface, mostrata nella Figura 17.11.

Figura 17.11 – Gli authentication scheme vengono aggiunti all'applicazione con il relativo metodo Add.

Per aggiungere l'autenticazione basata sui cookie, usiamo dunque l'extension method `AddCookie`, e impostiamo le sue opzioni come è mostrato dall'Esempio 17.6.

Esempio 17.6

```
public void ConfigureServices(IServiceCollection services)
{
   //Usiamo AddAuthentication per poi configurare gli scheme supportati
   services.AddAuthentication(
      defaultScheme: CookieAuthenticationDefaults.AuthenticationScheme
   )
   //Aggiungiamo lo scheme della cookie authentication
   .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options =>
   {
      //Modifichiamo qui le opzioni di emissione dei cookie
      options.ExpireTimeSpan = TimeSpan.FromDays(1);
   });
   //Qui registrazione di altri servizi
}
```

Dato che non stiamo usando ASP.NET Core Identity, spetta a noi l'onere di realizzare la UI di login e scrivere il codice per validare le credenziali dell'utente. Dopo aver verificato che username e password sono corretti, emettiamo il cookie di autenticazione grazie alla API `HttpContext.SignInAsync`. La vediamo nell'Esempio 17.7, che presenta un'action di login personalizzato in un'applicazione ASP.NET Core MVC.

Esempio 17.7

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel login, string redirectUrl)
{
   //Verifichiamo se esiste un utente con le credenziali fornite
   //Il metodo GetUserByCredentials contiene la logica personalizzata
   //per tale verifica. Lo user è anch'esso un nostro oggetto personalizzato
   var user = await GetUserByCredentials(login.Username, login.Password);
   if (user == null) {
      //Nessun utente trovato, vuol dire che le credenziali non erano valide
      ViewBag.Error = "Credentials are not valid!";
      return View();
   }
   //Creiamo una ClaimsIdentity e aggiungiamo i claim dell'utente loggato
   var claimsIdentity = new ClaimsIdentity(
      authenticationType: CookieAuthenticationDefaults.AuthenticationScheme
   );
   claimsIdentity.AddClaim(new Claim(ClaimTypes.Name, user.Username));
   claimsIdentity.AddClaim(new Claim(ClaimTypes.Role, user.Role));
   //Incapsuliamo tutto in un ClaimsPrincipal
   var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
   //Emettiamo il cookie di autenticazione fornendo delle opzioni
   var authProperties = new AuthenticationProperties {
      //Se l'utente vuole restare loggato, il cookie è persistente
      IsPersistent = login.RememberMe
   };
   await HttpContext.SignInAsync(
      scheme: CookieAuthenticationDefaults.AuthenticationScheme,
      principal: claimsPrincipal,
      properties: authProperties);
   //Login effettuato: indirizziamo l'utente alla pagina di provenienza
   return Redirect(redirectUrl ?? "/");
}
```

Anche in questo caso, le impostazioni sulla durata avranno effetto solo emettendo un cookie persistente. A tal proposito, deve essere valorizzata la proprietà `IsPersistent` dell'oggetto `AuthenticationProperties`, da fornire come parametro di `HttpContext.SignInAsync`.

Per effettuare il logout, usiamo l'extension method `HttpContext.SignOutAsync` fornendo il nome dello scheme, come viene indicato nell'Esempio 17.8.

Esempio 17.8

```
public async Task<IActionResult> Logout() {
   await HttpContext.SignOutAsync(
      scheme: CookieAuthenticationDefaults.AuthenticationScheme
   );
   return Redirect("/");
}
```

Una dimostrazione di autenticazione basata sui cookie con login personalizzato è allegata al presente capitolo con il nome di `custom-cookie-auth`.

Autenticazione su token JWT

Anche quando creiamo delle Single-Page Application, delle app mobile o delle applicazioni desktop, si pone il problema di far autenticare gli utenti. Ricorrere ai cookie non si rivelerà il sistema più pratico, soprattutto se l'applicazione non gira in un browser. Come soluzione alternativa potremmo valutare l'uso di OAuth 2.0 e OpenID Connect ma ricorrere a questi protocolli non è strettamente necessario se l'applicazione client usa un backend ASP.NET Core Web API che mantiene i dati degli utenti in un proprio database locale. Infatti, se non necessitiamo di un identity provider esterno, allora l'alternativa meno complessa consiste nell'emettere semplici token JWT, scambiati tra server e client.

Un token JWT, proprio come un cookie, contiene l'identità dell'utente e gli permette di autenticarsi a ogni richiesta. A differenza dei cookie, però, sono ideali per essere usati con una Web API: un token JWT, infatti, può essere facilmente inviato tale e quale via query string, in un'intestazione o nel corpo della richiesta HTTP, a discrezione dello sviluppatore che realizza l'applicazione backend.

JWT (JSON Web Token) è uno standard aperto che definisce le regole per creare un token di accesso che contiene i claim dell'utente ed è firmato digitalmente con una chiave segreta. Una descrizione tecnica approfondita delle peculiarità dei token JWT si trova nel sito di riferimento, all'indirizzo: `http://aspit.co/bnt`.

Per configurare l'autenticazione con token JWT nella nostra applicazione ASP.NET Core Web API, usiamo l'extension method `AddJwtBearer` come nell'Esempio 17.9.

Esempio 17.9

```
services.AddAuthentication(opts =>
{
   opts.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(opts =>
{
   opts.TokenValidationParameters = new TokenValidationParameters
   {
      ValidateIssuer = true,
      ValidateAudience = true,
      ValidateIssuerSigningKey = true,
      ValidIssuer = "Issuer",
      ValidAudience = "Audience",
      IssuerSigningKey = new SymmetricSecurityKey(
         Encoding.UTF8.GetBytes("SecretKey")
      ),
      //Tolleranza sulla data di scadenza del token
      ClockSkew = TimeSpan.Zero
   };
});
```

Il prossimo esempio mostra invece un metodo di creazione del token, da invocare all'atto del login, dopo aver verificato le credenziali dell'utente e aver ottenuto i suoi claim dal database. Il token così creato, potrà poi essere inviato al client nella risposta HTTP da un'action o con un middleware.

Esempio 17.10

```
private string CreateTokenForIdentity(IEnumerable<Claim> claims)
{
   var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("SecretKey"));
   var cred = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
   var token = new JwtSecurityToken(
      issuer: "Issuer",
      audience: "Audience",
      claims: claims,
      expires: DateTime.Now.AddMinutes(20),
      signingCredentials: cred
```

```
    );
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

In ogni applicazione ASP.NET Core è possibile configurare più di un authentication scheme usando gli extension method `Add` in successione. Così, offriremo agli utenti molteplici opzioni per autenticarsi, come nell'applicazione di esempio `ui-and-webapi-auth` allegata al presente capitolo.

Configurare l'autenticazione: identità federata

Mantenere un database di account locali è una scelta comune ma non priva di difetti. Dobbiamo considerare che, dal punto di vista di un utente, registrarsi può rappresentare una barriera all'utilizzo dell'applicazione, sia per il tempo richiesto sia perché non è detto che egli voglia digitare tutte le informazioni richieste. In altre situazioni, gestire account locali è del tutto precluso dai requisiti di progetto, come nel caso in cui l'applicazione debba inserirsi in un contesto di Single Sign-On.

ASP.NET Core accontenta sia le esigenze degli utenti consumer sia quelle dei membri di un'organizzazione, supportando sia il login con Microsoft Account e con i social network di Facebook, Google e Twitter, sia ogni tipo di identity provider esterno mediante i protocolli OAuth 2.0 e OpenID Connect.

Come vedremo fra poco, ASP.NET Core rende facile delegare la fase di login a un identity provider, che si farà carico di verificare le credenziali e di restituire un token crittograficamente sicuro contenente l'identità dell'utente. Le questioni legate alla sicurezza, come lo scambio dati e la verifica dell'integrità del token, sono gestite dal middleware di autenticazione di ASP.NET Core.

Login con i social network

Supportare il login con i social è un ottimo modo per agevolare gli utenti consumer nell'utilizzo della nostra applicazione, dato che potranno identificarsi senza sforzo e senza previa registrazione. Affinché i social possano accettare richieste di login dalla nostra applicazione, è necessario iscriverla per ottenere un identificativo e una chiave segreta da indicare come opzioni, come si può vedere nell'Esempio 17.11. Le istruzioni per ottenere tali valori da ogni social sono riportate nella documentazione ufficiale, raggiungibile all'indirizzo: `http://aspit.co/bne`.

Esempio 17.11

```
services.AddAuthentication()
.AddMicrosoftAccount(options => {
   options.ClientId = "myid";
   options.ClientSecret = "mysecret";
})
.AddFacebook(options => {
   options.AppId = "myid";
   options.AppSecret = "mysecret";
})
.AddGoogle(options => {
   options.ClientId = "myid";
   options.ClientSecret = "mysecret";
})
.AddTwitter(options => {
   options.ConsumerKey = "mykey";
   options.ConsumerSecret = "mysecret";
});
```

Il login con i social non sostituisce l'accesso con account locale, per cui le due modalità possono convivere nella stessa applicazione. Se stiamo usando ASP.NET Core Identity, la sua UI mostrerà automaticamente entrambe le possibilità di accesso nella pagina di login, come nella Figura 17.12.

Figura 17.12 – Opzioni di login con account locale e con i social nella UI di ASP.NET Core Identity.

Cliccando uno dei bottoni social, l'utente è reindirizzato alla pagina di login sul social network, per poi tornare nelle pagine dell'applicazione come utente autenticato. In questa fase, è possibile richiedere al social network di inviarci un certo numero di informazioni dal profilo dell'utente, come il suo nome, la data di nascita o la località di residenza, ovviamente con l'assenso dell'utente.

Login su provider esterni con OAuth 2.0 e OpenID Connect

Il supporto agli identity provider esterni non si ferma ovviamente solo ai servizi social, dato che il middleware di autenticazione di ASP.NET Core è in grado di interagire con qualsiasi altro tipo di identity provider che supporti il protocollo OAuth 2.0 o OpenID Connect. L'Esempio 17.12 mostra l'utilizzo degli extension method `AddOAuth` e `AddOpenIdConnect` ma è necessario completare la configurazione, indicando le informazioni specifiche fornite dal gestore dell'identity provider.

Esempio 17.12

```
services.AddAuthentication()
.AddOAuth("SchemeName", options => {
   //Configurazione specifica
})
.AddOpenIdConnect(options => {
   //Configurazione specifica
});
```

Come guida all'utilizzo degli extension method `AddOAuth` *e* `AddOpenIdConnect`*, Microsoft ha pubblicato delle applicazioni dimostrative nel repository GitHub di ASP.NET Core Identity, raggiungibile all'indirizzo: http://aspit.co/bnh. In particolar modo, sono rilevanti le classi Startup degli esempi SocialSample e OpenIdConnectSample per avere una panoramica dei valori di configurazione coinvolti.*

Quando ci troviamo a gestire svariate applicazioni per una stessa organizzazione, diventa importante costruire un sistema di Single Sign-On. Questo compito è reso semplice da IdentityServer4, un progetto open source di terze parti per realizzare un identity provider con ASP.NET Core. Dato che offre il supporto ai protocolli OAuth 2.0 e OpenID Connect, è interoperabile con applicazioni sviluppate in altri linguaggi. La documentazione è raggiungibile da `http://aspit.co/bni`.

Autenticazione Windows

Per applicazioni destinate alla intranet aziendale, possiamo valutare l'uso dell'autenticazione Windows, che permette agli utenti di essere identificati con l'account con cui hanno eseguito l'accesso al loro PC. Questo tipo di autenticazione è supportata in ASP.NET Core usando il web server HTTP.sys e perciò richiede il deploy dell'applicazione su una macchina Windows Server. L'Esempio 17.13 mostra come configurare il web host dalla classe `Program`.

Esempio 17.13

```
public static IWebHost BuildWebHost(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
   .UseHttpSys(options =>
   {
   options.Authentication.Schemes =
      AuthenticationSchemes.NTLM | AuthenticationSchemes.Negotiate;
   //Se vogliamo che l'utente risulti loggato in qualsiasi pagina
   //options.Authentication.AllowAnonymous = false;
   })
   .UseStartup<Startup>()
   .Build();
```

Poi, impostiamo come default il meccanismo di autenticazione esposto da HTTP.sys, usando l'extension method `AddAuthentication` come si può vedere nell'Esempio 17.14.

Esempio 17.14

```
services.AddAuthentication(HttpSysDefaults.AuthenticationScheme);
```

Quando un utente visita un'applicazione ASP.NET Core configurata in questo modo, risulterà automaticamente loggato purché stia usando Internet Explorer o Google Chrome. Questi browser sono pre-configurati in maniera tale da consentire il pass-through dell'autenticazione, ovvero inoltrare all'applicazione l'identità dell'utente attualmente loggato su Windows, in maniera automatica e senza inserimento di credenziali. Questo comportamento si verifica solo in ambiente locale, ovvero quando l'applicazione web è esposta con un nome host appartenente alla intranet aziendale, come per esempio, `http://nomemacchina`. Altri nomi a dominio possono essere aggiunti manualmente, per abilitare questa funzionalità anche in applicazioni pubblicate su rete internet. I browser Mozilla Firefox e Microsoft Edge chiederanno all'utente di digitare le credenziali ma Firefox può anche essere configurato per abilitare il pass-through dell'autenticazione automaticamente.

Quando un utente invia una richiesta HTTP, ASP.NET Core la autenticherà creando un'identità di tipo `WindowsIdentity`. In questa classe, derivata da `ClaimsIdentity`, troveremo dei claim che rappresentano gli attributi posseduti dall'utente Windows. L'applicazione può ovviamente operare delle logiche di autorizzazione in base a tali claim ma è bene tenere a mente che ASP.NET Core non impersonerà l'utente durante l'elaborazione della richiesta. Possiamo tuttavia eseguire porzioni di codice usando i privilegi dell'utente con il metodo `RunImpersonated`, come nell'Esempio 17.15.

Esempio 17.15

```
var identity = HttpContext.User.Identity as WindowsIdentity;
WindowsIdentity.RunImpersonated(identity.AccessToken, () =>
{
   //Qui accesso a risorse con privilegi dell'utente: es. lettura di un file
});
```

Questa tecnica è pensata per eseguire piccoli blocchi di codice e non deve assolutamente essere usata per eseguire interi middleware nel suo contesto.

Autenticazione con Azure Active Directory

Con la crescente digitalizzazione delle imprese, il raggiungimento degli obiettivi è influenzato in maniera determinante dalla qualità e dall'efficienza delle applicazioni web che supportano il business. Che si tratti di gestionali o di altro tipo di applicazioni, il cloud permette di ottenere l'alta disponibilità, la ridondanza geografica e la scalabilità che permettono a un'impresa di operare senza discontinuità. La migrazione al cloud può avvenire gradualmente grazie a servizi come Azure Active Directory che accorciano la distanza con l'infrastruttura on-premise e rendono possibili scenari ibridi.

Azure AD è un servizio d'identità ospitato nel cloud che può essere usato come estensione di una foresta di Active Directory on-premise. L'amministratore IT installa e configura un agent, denominato Azure AD Connect, che si occupa di sincronizzare nel cloud le identità degli utenti in maniera sicura. È possibile indicare selettivamente quali unità organizzative, utenti e attributi sincronizzare, in modo da trasferire solo le informazioni necessarie. Così, gli utenti potranno accedere alle applicazioni anche al di fuori del perimetro dell'organizzazione: il supporto alla multi-factor authentication e gli algoritmi di identity protection riducono drasticamente il rischio di furto degli account.

Azure AD può funzionare anche autonomamente: con l'interfaccia web del portale di Azure, l'amministratore può creare una directory simile a quella gestita on-premise da un domain controller. Tra i servizi offerti abbiamo una gestione completa delle identità, con gestione dei gruppi, dei dispositivi e delle applicazioni. Queste ultime sono facilmente abilitabili anche da un corposo catalogo contenente Office 365, Salesforce e Dropbox, tra le altre. Il tutto è corredato da una ricca suite di logging e auditing che permette all'amministratore di tenere sotto controllo l'attività svolta nella directory. Gli utenti, invece, possono contare su un servizio di cambio password self-service.

Per iniziare a usare Azure Active Directory come authentication scheme, rechiamoci nel portale di Azure, creiamo una nuova directory e registriamo la nostra applicazione come nella Figura 17.13.

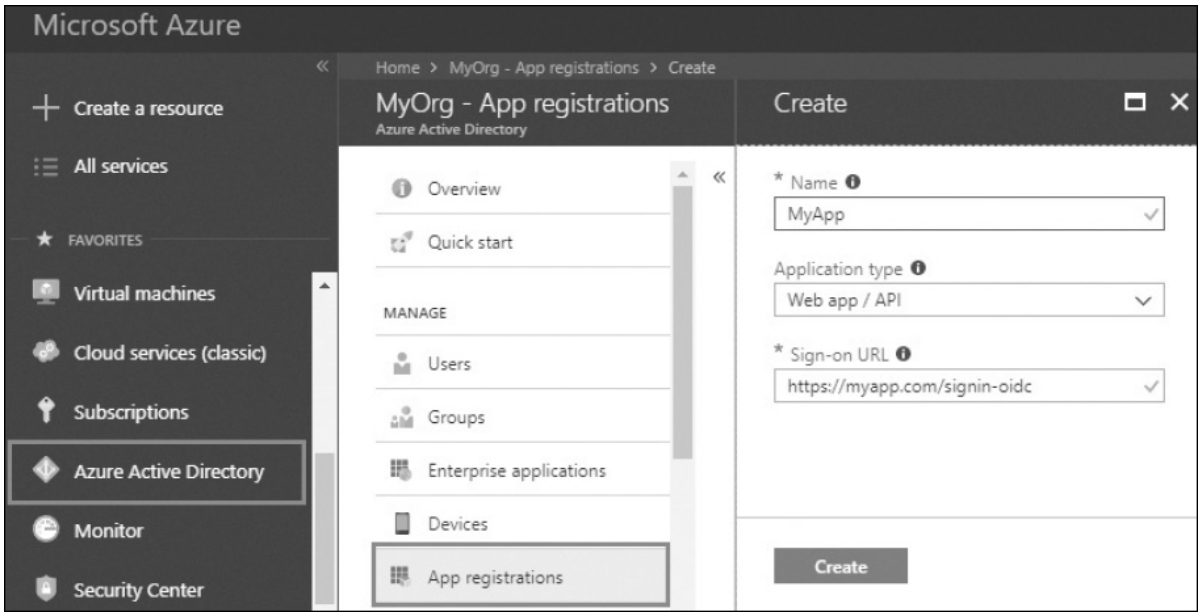


Figura 17.13 – Dopo aver creato una nuova Azure Active Directory, registriamo la nostra applicazione.

L'applicazione sarà abilitata a usare la directory per il login. Ora recuperiamo i seguenti valori:

- TenantId, ottenuto selezionando “Endpoints” dalla blade “App registrations” e selezionando il GUID che appare in uno qualsiasi degli URL in elenco.

- ClientId, è l’id dell’applicazione registrata, ottenuto dalla blade “App registrations” e selezionando “All apps”.

Tali valori dovranno essere usati con l’extension method `AddAzureAd`, come viene illustrato nell’Esempio 17.16. Non dimentichiamo di inserire anche il dominio della directory.

Esempio 17.16

```
services.AddAuthentication(opts => {
   opts.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
   opts.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
})
.AddAzureAd(opts => {
   opts.Instance = ”https://login.microsoftonline.com/”;
   opts.Domain = ”myorg.onmicrosoft.com”;
   opts.TenantId = ”aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee”;
   opts.ClientId = ”11111111-2222-3333-4444-555555555555”;
   opts.CallbackPath = ”/signin-oidc”;
})
.AddCookie();
```

Per avviare la procedura di login, prepariamo un’action come quella illustrata nell’Esempio 17.17.

Esempio 17.17

```
[HttpGet]
public IActionResult SignIn()
{
   //Url di ridirezione a login avvenuto
   var redirectUrl = Url.Action(”Index”, ”Home”);
   return Challenge(
      new AuthenticationProperties { RedirectUri = redirectUrl },
      OpenIdConnectDefaults.AuthenticationScheme);
}
```

Per l’approfondimento, si veda l’esempio di Microsoft al seguente indirizzo: http://aspit.co/bnj.

Se stiamo realizzando un’applicazione multi-tenant e intendiamo distribuirla a varie organizzazioni con il modello SaaS (Software as a Service), si rende necessario supportare molteplici istanze di Azure AD. Questo approccio, del tutto simile a quello usato da Office 365, è dimostrato in questa applicazione di esempio pubblicata da Microsoft: http://aspit.co/bnk.

Autorizzazione in ASP.NET Core MVC e Web API

Dal momento che il middleware di autenticazione di ASP.NET Core ci ha consentito di identificare l’utente, ora possiamo passare alla fase di autorizzazione, momento in cui verifichiamo se l’identità e i claim contenuti in essa sono adeguati per consentire l’accesso alla risorsa richiesta.

Questa fase può essere svolta con un middleware personalizzato ma, se stiamo sviluppando un’applicazione ASP.NET Core MVC o Web API, è preferibile sfruttare l’apposito `AuthorizeAttribute`.

L’attributo AuthorizeAttribute

Come in precedenti versioni di ASP.NET MVC, anche in ASP.NET Core MVC possiamo decorare le action con l’AuthorizeAttribute per impedire l’accesso agli utenti anonimi, cioè a coloro che non risultano autenticati. Nel caso in cui volessimo proteggere tutte le action di un controller, possiamo porre l’`AuthorizeAttribute` sul controller stesso, come viene indicato nell’Esempio 17.18.

Esempio 17.18

```
[Authorize]
public class AccountController : Controller
{
   //Qui sono definite le action del controller
}
```

ASP.NET Core MVC ci dà anche la possibilità di proteggere tutti i controller dell’intera applicazione, aggiungendo un apposito `AuthorizeFilter` a livello globale. L’Esempio 17.19 mostra come configurarlo dall’extension method `AddMvc`, in `ConfigureServices` della classe `Startup`.

Esempio 17.19

```
services.AddMvc(options => {
   var policy = new AuthorizationPolicyBuilder()
                        .RequireAuthenticatedUser()
                        .Build();
   var filter = new AuthorizeFilter(policy);
   //Ogni controller e ogni action saranno inaccessibili agli utenti anonimi
   options.Filters.Add(filter);
});
```

Questa impostazione è particolarmente consigliata quando realizziamo web application o web API che richiedono l'autenticazione per tutte le action, tranne quella di login.

L'attributo AllowAnonymousAttribute

Quando proteggiamo un controller o l'intera applicazione, è importante poi riabilitare l'accesso agli utenti anonimi almeno per l'action di login, in modo che possano accedervi per fornire le proprie credenziali. Nell'Esempio 17.20, facciamo opt-out dal meccanismo di autorizzazione, usando l' `AllowAnonymousAttribute` sulle action di login, che così diventeranno pubblicamente accessibili.

Esempio 17.20

```
[Authorize]
public class AccountController : Controller {
   [HttpGet, AllowAnonymous]
   public async Task<IActionResult> Login() {
      // Mostriamo la view di login all'utente
   }
   [HttpPost, AllowAnonymous]
   public async Task<IActionResult> Login(LoginViewModel model) {
      // Qui verifichiamo le credenziali fornite
   }
   //Qui altre action protette
}
```

Le altre eventuali action definite nel controller continueranno a essere protette, data la presenza dell'`AuthorizeAttribute` sul controller stesso.

Autorizzazione in base al ruolo

L'`AuthorizeAttribute` è molto versatile perché ci consente di definire in maniera molto precisa i requisiti che l'utente deve possedere affinché possa accedere ai controller e alle action. In determinate situazioni, infatti, non è sufficiente che l'utente sia autenticato ma deve anche possedere il ruolo per compiere una determinata operazione. È una prassi comune quella di assegnare ruoli specifici ai vari utenti, in modo da abilitarli alle sole operazioni di loro competenza. Nell'Esempio 17.21, usiamo l'`AuthorizeAttribute` e la sua proprietà `Roles` per indicare uno o più ruoli separati da virgola. Soltanto gli utenti appartenenti a uno dei ruoli indicati potranno avere accesso all'action `IndexAll`.

Esempio 17.21

```
[Authorize]
public class CustomerController : Controller {
   [Authorize(Roles = "Administrator,PowerUser")]
   public async Task<IActionResult> IndexAll() {
      //Solo gli utenti con ruolo Administrator o con ruolo PowerUser
      //potranno visualizzare l'elenco completo dei clienti
   }
}
```

Con ASP.NET Core, il ruolo di un utente è un claim come qualsiasi altro, da aggiungere una o più volte alla `ClaimsIdentity` con il tipo `ClaimTypes.Role`.

Usare le policy per criteri avanzati di autorizzazione

ASP.NET Core MVC introduce il supporto alle policy, che si rivelano utili nel momento in cui dobbiamo formulare logiche di autorizzazione avanzate o composte di vari criteri. Il caso più comune consiste nel definire una policy che verifichi la presenza di un determinato claim. L'Esempio 17.22 mostra la definizione di tale policy mediante l'extension method `AddAuthorization`, all'interno del metodo `ConfigureServices` della classe `Startup`.

Esempio 17.22

```
services.AddAuthorization(options => {
   options.AddPolicy("JustForCustomer1", policy => {
      //Imponiamo la presenza del claim Email
      policy.RequireClaim(ClaimTypes.Email);
      //Imponiamo che il claim "Tenant" sia uguale a "Customer1"
      policy.RequireClaim("Tenant", "Customer1");
   });
});

//Qui registrazione di altri servizi
```

Grazie al metodo `RequireClaim`, possiamo configurare la policy in modo da richiedere la presenza di un determinato claim o imporre che assuma un determinato valore. In aggiunta, abbiamo a disposizione i metodi specializzati `RequireRole` e `RequireUserName` per verificare i claim del ruolo e del nome utente. Con `RequireAuthenticatedUser` richiediamo che l'utente sia semplicemente autenticato. Una volta preparata la policy, possiamo usarla indicandone il nome nella proprietà `Policy` dell'`AuthorizeAttribute`, come viene illustrato nell'Esempio 17.23.

Esempio 17.23

```
public class ReportController : Controller
{
   [Authorize(Policy = "JustForCustomer1")]
   public async Task<IActionResult> SendCustomizedReportViaEmail() {
      //Invia un report all'indirizzo email dell'utente
   }
}
```

Se i claim posseduti dall'utente non fossero conformi a quelli indicati nella policy, l'applicazione ASP.NET Core MVC mostrerà un messaggio di accesso non autorizzato.

Policy con logica di autorizzazione personalizzata

Quando la verifica dei claim non è sufficiente, possiamo attuare logiche di autorizzazione personalizzata aggiungendo uno o più requirement alla policy, come mostrato nell'Esempio 17.24.

Esempio 17.24

```
services.AddAuthorization(options => {
   options.AddPolicy("VipCustomers", policy => {
      //Aggiungiamo il requirement di un importo minimo acquistato
      policy.Requirements.Add(
         new MinimumPurchaseRequirement(minimum: 1000)
      );
   });
});
```

Un requirement contiene dei parametri di validità come, per esempio, l'importo minimo speso da un utente affinché possa essere considerato un cliente VIP. Tipicamente, informazioni come questa non vengono inserite tra i claim della `ClaimsIdentity` perché potrebbero cambiare durante il nomale utilizzo dell'applicazione da parte dell'utente. Per questo motivo abbiamo bisogno di creare un requirement, ovvero una nostra classe personalizzata a cui facciamo implementare l'interfaccia `IAuthorizationRequirement`, come nell'Esempio 17.25.

Esempio 17.25

```
public class MinimumPurchaseRequirement : IAuthorizationRequirement
{
   public MinimumPurchaseRequirement(decimal minimum) {
      Minimum = minimum;
   }
   public decimal Minimum { get; }
}
```

Possiamo riutilizzare la classe del requirement in diverse policy, eventualmente creandone varie istanze a cui vengono forniti valori differenti. A questo punto, occorre creare una seconda classe derivante da `AuthorizationHandler<TRequirement>` che conterrà la logica di autorizzazione vera e propria, come si può vedere nell'Esempio 17.26.

Esempio 17.26

```
public class MinimumPurchaseAuthorizationHandler :
   AuthorizationHandler<MinimumPurchaseRequirement>
{
   protected override async Task HandleRequirementAsync(
      AuthorizationHandlerContext context,
      MinimumPurchaseRequirement requirement)
   {
      //Recuperiamo dal database il valore speso fino a oggi dall'utente
      decimal purchased = await GetAmount(context.User.Identity.Name);
      //Lo confrontiamo con quello minimo indicato nel requirement
      if (purchased < requirement.Minimum)
      {
         //Non ha raggiunto il minimo, l'autorizzazione fallisce
         context.Fail();
      }
      //Altrimenti, l'autorizzazione ha successo
      context.Succeed(requirement);
   }
}
```

All'interno dell'override di `HandleRequirementAsync` invochiamo i metodi `Succeed` o `Fail` dell'oggetto `AuthorizationHandlerContext` per segnalare l'esito della verifica.

Gli `AuthorizationHandler` possono sfruttare il meccanismo della dependency injection di ASP.NET Core per ottenere un riferimento agli altri servizi configurati nell'applicazione, come, per esempio, il `DbContext` di Entity Framework Core. Tuttavia, interrogare il database a ogni esecuzione dell' `AuthorizationHandler` avrà un impatto sui tempi di risposta dell'applicazione e potrebbe risultare superfluo. Le prestazioni miglioreranno sfruttando la cache o inserendo il valore come claim nella `ClaimsIdentity`, purché si possa tollerare la presenza di dati potenzialmente obsoleti.

Come si vede nell'Esempio 17.27, non resta che registrare il nostro authorization handler come servizio dell'applicazione, conferendogli il ciclo di vita che riteniamo opportuno usare.

Esempio 17.27

```
public void ConfigureServices(IServiceCollection services) {
   services.AddScoped<IAuthorizationHandler,
            MinimumPurchaseAuthorizationHandler>();
}
```

In questo esempio è stato usato `AddScoped` per generare una nuova istanza a ogni richiesta HTTP. Anche `AddSingleton` è un'opzione percorribile, purché l'authorization handler non mantenga un suo stato interno e l'istanza sia perciò riutilizzabile da ogni richiesta.

Autorizzazione in base all'authentication scheme

Se stiamo supportando più di un authentication scheme, possiamo fare in modo che un'action sia accessibile solo se si è autenticati con uno scheme in particolare. Nell'Esempio 17.28, vediamo l'`AuthorizeAttribute` usato con la sua proprietà `AuthenticationSchemes`.

Esempio 17.28

```
[Authorize(Roles = "Crm")]
public class SaleController : Controller {
   [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
   public async Task<IActionResult> GetRecentOrders() {
      //logica
   }
}
```

Da questo esempio possiamo notare come gli `AuthorizeAttribute` siano modulari e componibili tra loro: nella fattispecie, l'utente deve sia possedere il ruolo "Crm" sia essersi autenticato con token JWT. Questa componibilità diventa particolarmente importante per modellare ancor più precisamente i criteri di autorizzazione in scenari più complessi.

Autorizzazione on demand con IAutorizationService

L'`AuthorizeAttribute` svolge un compito eccellente nel separare la logica di autorizzazione dalla logica applicativa ma in rare occasioni potrebbe essere necessario fare l'autorizzazione in maniera imperativa, direttamente nel codice dell'action. In questa situazione possiamo anche sfruttare il model binding di ASP.NET Core MVC per prendere più facilmente decisioni in base ai dati contenuti nella richiesta. A questo scopo, usiamo il servizio `IAuthorizationService` e il suo metodo `AuthorizeAsync`, come viene mostrato nell'Esempio 17.29.

Esempio 17.29

```
public async Task<IActionResult> DeleteOrder(
   int orderId, [FromServices] IAuthorizationService authService)
{
   //Usiamo l'orderId che ci viene valorizzato dal Model Binder
   Order order = await ctx.Orders.FindAsync(orderId);
   //Forniamo l'utente, la risorsa da autorizzare e la policy
   AuthorizationResult result =
            await authService.AuthorizeAsync(User, order, "SenderOfOrder");
   if (result.Succeeded) {
      //Eliminiamo l'ordine ma solo se l'autorizzazione ha avuto successo
      ctx.Orders.Remove(order);
      await ctx.SaveChangesAsync();
   }
   return RedirectToAction(nameof(Index));
}
```

In allegato al capitolo è presente l'applicazione dimostrativa `customer-authorization`, che riepiloga tutte le modalità di autorizzazione presentate finora.

Conclusioni

In questo capitolo abbiamo imparato a configurare le funzionalità di autenticazione e autorizzazione. Queste responsabilità sono tutt'altro che banali da implementare ed è per questo motivo che ASP.NET Core ci supporta con un middleware robusto e funzionale, che è in grado di lavorare con tutti i principali meccanismi di autenticazione locale e federata. Inoltre, grazie ad ASP.NET Core Identity, anche i compiti più laboriosi e ripetitivi, come realizzare la UI di gestione dell'account, diventano possibili con poche righe di codice e una configurazione minimale.

Garantire un accesso sicuro è solo parte della storia: la nostra applicazione deve infatti assicurare che i dati forniti dagli utenti siano trattati in maniera confidenziale e siano resi accessibili soltanto da coloro che ne possiedono i privilegi. ASP.NET Core MVC offre un sistema chiaro e versatile per esplicitare le policy di autorizzazione direttamente a corredo del codice che intendiamo proteggere. Il tema della sicurezza merita un ulteriore approfondimento, ed è per questo che, nel Capitolo 18, ci addentreremo ulteriormente in ASP.NET Core Identity e nelle altre funzionalità di ASP.NET Core pensate appositamente per garantire la completa protezione delle informazioni sensibili.

18
Sicurezza nelle applicazioni ASP.NET Core

Costruire un'applicazione sicura è ormai un requisito imprescindibile. Gli utenti che utilizzano la nostra applicazione meritano che i loro dati siano trattati con la massima confidenzialità e trasparenza. Questo non comprende solo le informazioni sensibili ma ogni genere di dato, come le ricerche, le preferenze di acquisto e i messaggi scambiati con l'assistenza. Il diritto alla privacy non è solo una questione legislativa ma è l'elemento fondamentale su cui viene costruito il rapporto di fiducia tra gli utilizzatori e il servizio offerto dall'applicazione.

ASP.NET Core rende semplice proteggere i dati sia in-transit, ovvero durante lo scambio dati tra client e server che at-rest, ovvero mentre sono memorizzati sul disco o nel database. Inoltre, continuando la nostra esplorazione di ASP.NET Core Identity, scopriremo le funzionalità che offre in merito alla protezione dell'account, come la two-factor authentication, ora sfruttabile in ogni applicazione per garantire agli utenti una procedura di login ancora più sicura. Infine, esamineremo gli attacchi HTTP più comuni e capiremo quali contromisure possiamo adottare per difendere l'applicazione.

Personalizzare l'account dell'utente

Come abbiamo visto nel capitolo precedente, iniziare a usare ASP.NET Core Identity è molto semplice perché abbiamo a disposizione un template di progetto già pronto. Tuttavia, è anche probabile che dovremo apportare modifiche all'account dell'utente, in modo che includa altre proprietà – come il codice fiscale o la residenza – oltre a quelle esistenti. ASP.NET Core Identity è molto estendibile ed è importante imparare a conoscere i suoi punti di estendibilità anziché creare un sistema di membership personalizzato, che non adotterebbe tutti gli accorgimenti in merito alla sicurezza.

Aggiungere proprietà personalizzate al profilo dell'utente

La personalizzazione di gran lunga più frequente consiste nell'aggiungere altre proprietà a quelle già presenti in `IdentityUser`, che è la classe base fornita da ASP.NET Core Identity per rappresentare il profilo di un utente. Possiamo estenderla creando una classe personalizzata, chiamata per esempio `ApplicationUser`, come è mostrato nell'Esempio 18.1.

Esempio 18.1

```
public class ApplicationUser : IdentityUser
{
   //Aggiungiamo una proprietà personalizzata
   public string FiscalCode { get; set; }
}
```

Nella classe `ApplicationUser` possiamo ovviamente definire sia proprietà scalari sia proprietà di navigazione verso altre entità correlate. Se stiamo usando Entity Framework Core per la persistenza degli account, allora dobbiamo anche creare una classe `ApplicationDbContext` che facciamo derivare da `IdentityContext<TUser>` per ridefinire le regole di mapping predefinite o per aggiungerne di nuove. L'Esempio 18.2 mostra come creare tale classe.

Esempio 18.2

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
   //Eventuale mapping aggiuntivo nel metodo OnModelCreating
}
```

A fronte di modifiche al mapping dovremo anche aggiornare lo schema del database manualmente oppure con un'apposita code migration, come si può vedere nell'Esempio 18.3.

Esempio 18.3

```
dotnet ef migrations add "FiscalCode added"
dotnet ef database update
```

Ora aggiorniamo la configurazione di ASP.NET Core Identity dal metodo `ConfigureServices` della classe `Startup`, come si può vedere nell'Esempio 18.4.

Esempio 18.4

```
services.AddDefaultIdentity<ApplicationUser>()
           .AddEntityFrameworkStores<ApplicationDbContext>();
```

Infine, se nell'applicazione stiamo usando la classe `UserManager<IdentityUser>` per accedere al database locale, allora al suo posto dobbiamo usare `UserManager<ApplicationUser>`. Ora che abbiamo personalizzato la classe `ApplicationUser`, dobbiamo anche consentire agli utenti di modificare i nuovi valori che sono stati aggiunti al profilo.

Personalizzare la pagina di registrazione

A partire da ASP.NET Core Identity 2.1, le pagine di gestione del profilo non si trovano più nel progetto ma vengono referenziate come pacchetto NuGet. Questo approccio, definito UI as a library, presenta i suoi vantaggi: se Microsoft dovesse rilasciare degli aggiornamenti di sicurezza, sarà possibile ottenerli semplicemente aggiornando il pacchetto. L'aspetto grafico delle pagine è comunque personalizzabile con regole di stile CSS ma, per modifiche strutturali, è necessario usare la funzionalità di scaffolding per reintrodurre le pagine nel progetto, in modo da poterle modificare. Per fare questo, da Visual Studio facciamo tasto destro sul progetto e poi clicchiamo il menù `Add -> New scaffolded item -> Identity` per accedere al wizard di selezione delle pagine, come si può vedere nella Figura 18.1.

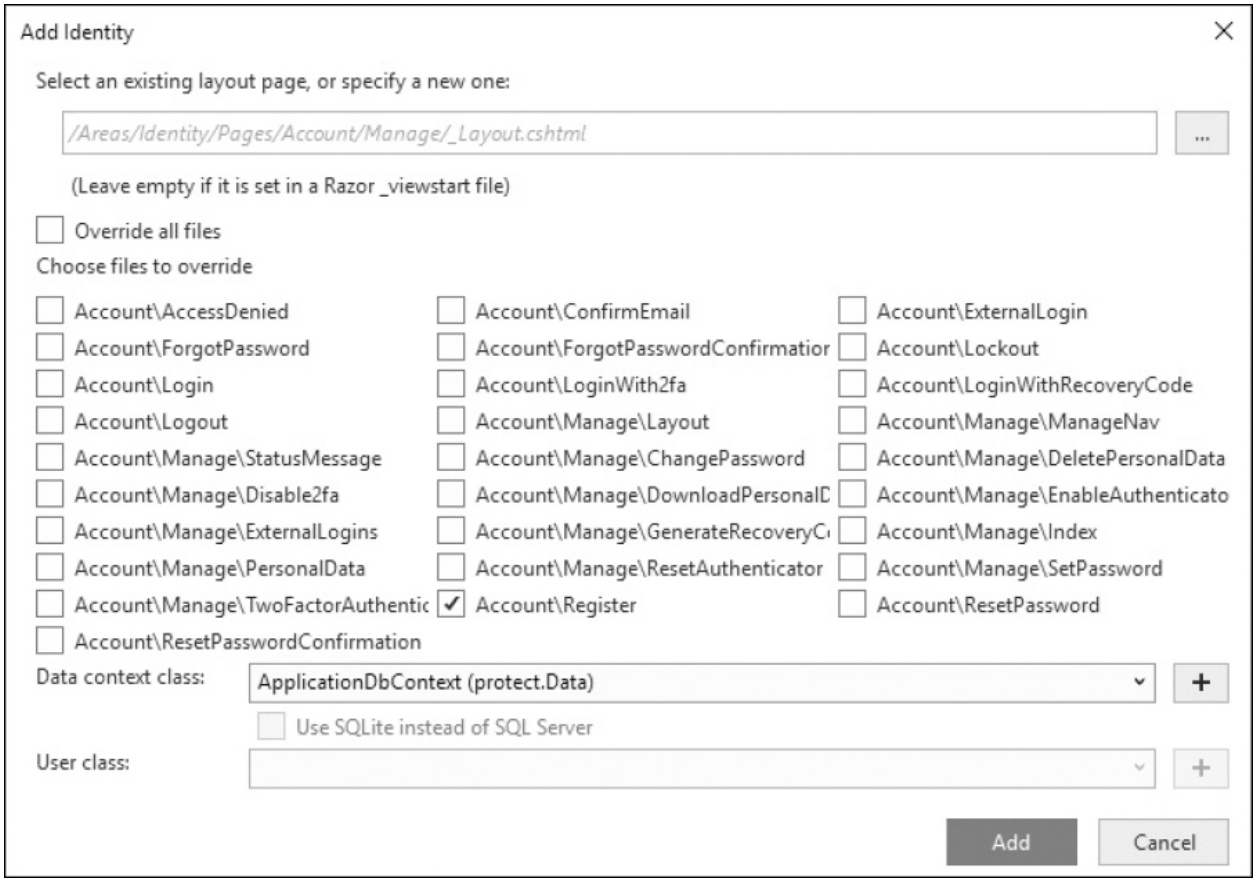


Figura 18.1 – Con la funzionalità di scaffolding aggiungiamo al progetto le pagine da personalizzare.

Le pagine selezionate dal wizard vengono copiate nella directory/Areas/Identity/Pages/Account del progetto. Si tratta di Razor Pages con estensione `.cshtml`, che quindi dispongono di un relativo codefile `.cshtml.cs`. Apriamo la pagina `Register.cshtml` e aggiungiamo all'interno del form il codice necessario a consentire all'utente di digitare il codice fiscale, come si può vedere nell'Esempio 18.5.

Esempio 18.5

```
<div class="form-group">
   <label asp-for="Input.FiscalCode"></label>
   <input asp-for="Input.FiscalCode" class="form-control" />
   <span asp-validation-for="Input.FiscalCode" class="text-danger"></span>
</div>
```

Ora apriamo il relativo codefile che si trova nella stessa directory e aggiungiamo la proprietà `FiscalCode` alla classe che agisce da model per la Razor Page, come si può vedere nell'Esempio 18.6.

Esempio 18.6

```
[Required]
[Display(Name = "Codice fiscale")]
public string FiscalCode { get; set; }
```

La data annotation `[Required]` rende il campo obbligatorio, in modo che l'utente debba fornirlo già dalla fase di registrazione. Quando l'utente invia il form di registrazione, dovremo copiare il valore della proprietà `FiscalCode` sull'oggetto `ApplicationUser` che

rappresenta il nuovo utente. Ancora una volta, possiamo farlo dal codefile, integrando il metodo `OnPostAsync` come si può vedere nell'Esempio 18.7.

Esempio 18.7

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
   if (ModelState.IsValid)
   {
      var user = new ApplicationUser
      {
         UserName = Input.Email,
         Email = Input.Email,
         //Copiamo il codice fiscale sull'ApplicationUser
         FiscalCode = Input.FiscalCode
      };
      var result = await _userManager.CreateAsync(user, Input.Password);
      //...
   }
   //...
}
```

In questo esempio abbiamo modificato solo la pagina di registrazione ma, ovviamente, dovremo dare modo all'utente di modificare il valore anche in seguito, dalla pagina di modifica dei dati personali. Per questo è necessario fare lo scaffolding anche della view `PersonalData.cshtml`.

Ora che abbiamo capito come personalizzare le view di gestione del profilo, andiamo a vedere quali sono i meccanismi di sicurezza di cui beneficiamo usando la UI di ASP.NET Core Identity.

Sicurezza degli account locali

ASP.NET Core Identity ci permette di indicare precisamente quanto devono essere stringenti i criteri di sicurezza da impiegare nelle fasi di registrazione e login dell'utente. La configurazione di default risulta già adeguata a gran parte delle applicazioni ma, negli scenari che richiedono un elevato grado di sicurezza, è necessario prestare attenzione alle funzionalità avanzate e imparare a farne un uso corretto per garantire ai nostri utenti il miglior grado di protezione del loro account.

ASP.NET Core Identity, per default, memorizza i dati personali dell'utente in chiaro nel database. La password, invece, passa attraverso un robusto algoritmo di key derivation e perciò ne viene salvato solo un hash. Per attuare la crittografia at-rest di tutti i dati personali, dobbiamo ricorrere alle funzionalità offerte dalla piattaforma in uso come la Transparent Data Encryption (TDE) di SQL Server. Dato che questi temi riguardano da vicino l'ambito sistemistico, non li tratteremo in questo libro. Per l'approfondimento, si vedano i contenuti che Microsoft ha dedicato a questo tema all'indirizzo: `http://aspit.co/bos`. Invece, passiamo ora a vedere come configurare dall'applicazione i meccanismi di sicurezza offerti da ASP.NET Core Identity, iniziando proprio dai criteri che regolano la creazione delle password.

Criteri di sicurezza della password

Per default, in fase di registrazione è richiesto l'inserimento di una password con lunghezza minima di 6 caratteri e che contenga almeno una lettera maiuscola, una minuscola, un numero e un simbolo. Se questi criteri non dovessero risultare adeguati rispetto alle proprie policy, è possibile intervenire sulle opzioni di ASP.NET Core Identity, come nell'Esempio 18.8, per configurare, per esempio, una password più lunga ma senza il requisito di numeri o simboli.

Esempio 18.8

```
services.AddDefaultIdentity<ApplicationUser>(options => {
   options.Password.RequiredLength = 10;
   options.Password.RequireDigit = false;
   options.Password.RequireNonAlphanumeric = false;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

Per verificare che l'impostazione abbia avuto effetto, proviamo a registrarci. Se la password digitata non è conforme ai requisiti, appariranno uno più errori di validazione, come illustrato nella Figura 18.2.

Prima di memorizzare le password nel database, ASP.NET Core Identity le elabora con la funzione di derivazione PBKDF2, che applica un salt e compie un'operazione di hashing per migliaia di iterazioni. Questo serve a scoraggiare gli attacchi brute force e dictionary, che così richiedono una smisurata quantità di risorse computazionali per poter risalire alla password originale. All'atto del login, la password digitata dall'utente subirà lo stesso procedimento e il risultato verrà confrontato con quello presente nel database. Questo lavoro di

protezione è svolto dal servizio `PasswordHasher<TUser>`, che eventualmente possiamo sostituire con una nostra implementazione, se volessimo usare un algoritmo differente. L'Esempio 18.9 mostra la riga di codice per eseguire la sostituzione.

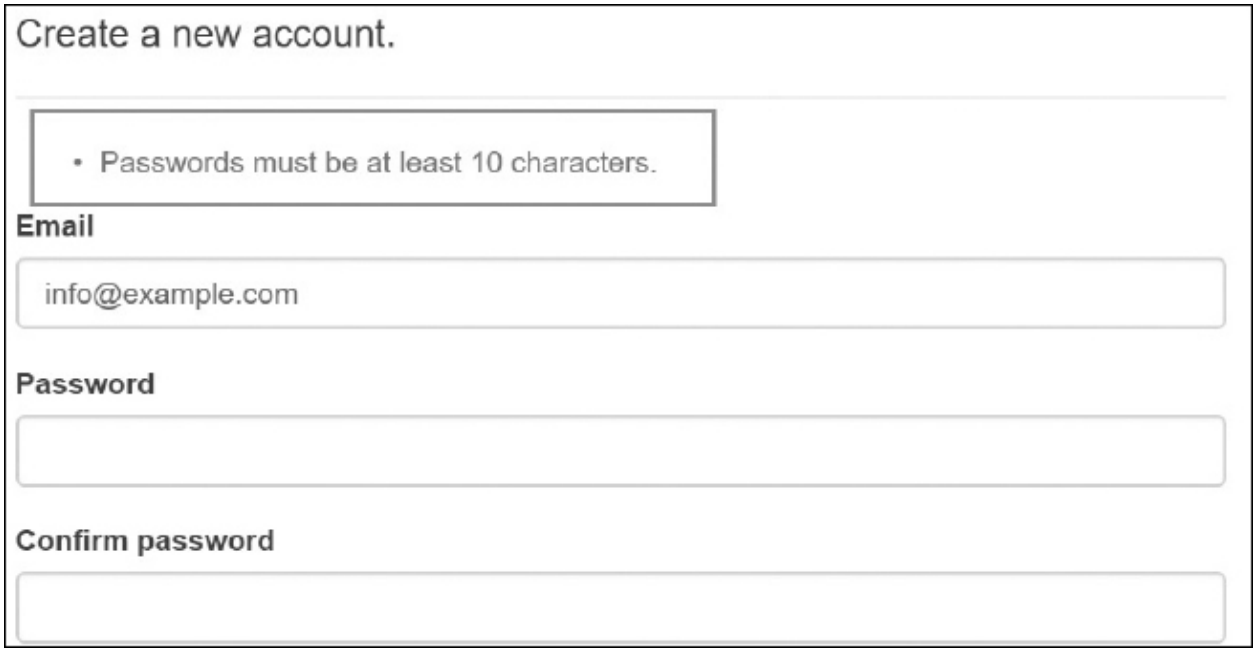


Figura 18.2 – La pagina di registrazione avvisa l'utente sui criteri di sicurezza della password.

Esempio 18.9

```
//MyHasher è una implementazione di IPasswordHasher<ApplicationUser>
services.AddTransient<IPasswordHasher<ApplicationUser>, MyHasher>();
```

Al termine della registrazione, l'utente risulterà immediatamente loggato e potrà navigare nelle aree riservate del sito. In alternativa, possiamo richiedere la conferma dell'e-mail prima che l'utente sia abilitato a fare il login.

Chiedere la conferma di registrazione

Se vogliamo un'ulteriore sicurezza che l'utente si sia registrato con un indirizzo e-mail che gli appartiene, è possibile fare in modo che gli venga inviato un link di conferma. L'utente potrà dunque confermare la sua registrazione e fare il login solo dopo aver cliccato il link. Nell'Esempio 18.10 configuriamo proprio questo requisito.

Esempio 18.10

```
services.AddDefaultIdentity<ApplicationUser>(options => {
   options.SignIn.RequireConfirmedEmail = true;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

Affinché ASP.NET Core Identity possa inviare l'email contenente il link, dobbiamo fornire un'implementazione per il servizio `IEmailSender`. In esso, inseriremo la logica di invio dell'e-mail con la classe `SmtpClient`, usando le impostazioni del nostro server SMTP. La registrazione del servizio è fatta come viene mostrato nell'Esempio 18.11.

Esempio 18.11

```
services.AddSingleton<IEmailSender, EmailSender>();
```

In questo modo l'utente riceverà l'email di conferma, come si vede nella Figura 18.3.

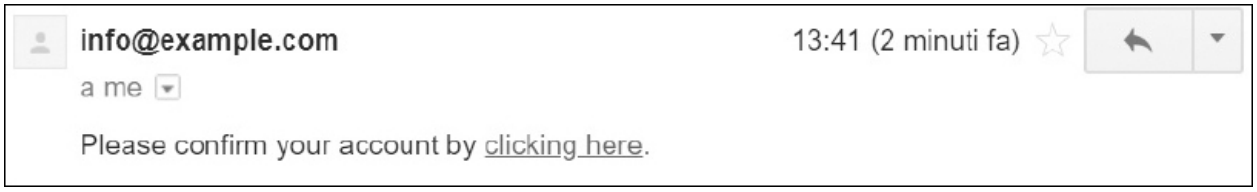


Figura 18.3 – Prima che l'utente possa accedere, possiamo richiedergli di cliccare un link in una e-mail.

Affinché l'intera procedura sia solida, dobbiamo evitare che ASP.NET Core Identity consideri l'utente come loggato subito dopo la sua registrazione iniziale. Per disabilitare questo comportamento, facciamo lo scaffolding della view di registrazione e dal file `Register.cshtml.cs` commentiamo la riga riportata nell'Esempio 18.12.

Esempio 18.12

```
//await signInManager.SignInAsync(user, isPersistent: false);
```

La conferma tramite e-mail è usata anche in altre situazioni come, per esempio, nella reimpostazione della password.

Reimpostazione della password

Come abbiamo visto in precedenza, le password sono memorizzate nel database solo dopo aver subito un processo crittografico che le rende irrecuperabili. Per questo motivo, se l'utente dovesse dimenticare la password, non potrà far altro che impostarne una nuova usando l'apposita procedura, accessibile dalla pagina di login, come si può vedere nella Figura 18.4.

Figura 18.4 – La maschera di login presenta anche un link per la reimpostazione della password.

Anche in questo caso, verrà recapitata un'e-mail contenente un link di conferma. Dopo averlo cliccato, l'utente potrà completare la procedura indicando una nuova password.

La sicurezza di queste procedure risiede nel fatto che il link contiene un codice crittograficamente sicuro, generato dal servizio `EmailTokenProvider`. Quando l'utente cliccherà il link, una pagina dell'applicazione riceverà il token e controllerà che corrisponda a quello emesso, a conferma che l'utente è il legittimo proprietario della casella e-mail che ha fornito.

ASP.NET Core Identity si avvale anche di un `AuthenticatorTokenProvider` per generare un token da usare per abilitare l'autenticazione a più fattori, come vedremo in seguito.

Two-factor authentication

Le applicazioni web che richiedono un elevato livello di sicurezza esigono che l'utente digiti una TOTP (time-based one time password), cioè un codice a tempo da inserire in aggiunta alla password di accesso. In questo modo, l'utente deve dimostrare sia di conoscere le credenziali (primo fattore) sia di possedere il dispositivo (secondo fattore) sul quale appare il codice a tempo. Grazie alla presenza di più fattori, è molto più difficile che un eventuale malintenzionato riesca a impossessarsi dell'account. L'utente può abilitare la two-factor authentication dalle pagine del profilo, come si può vedere nella Figura 18.5.

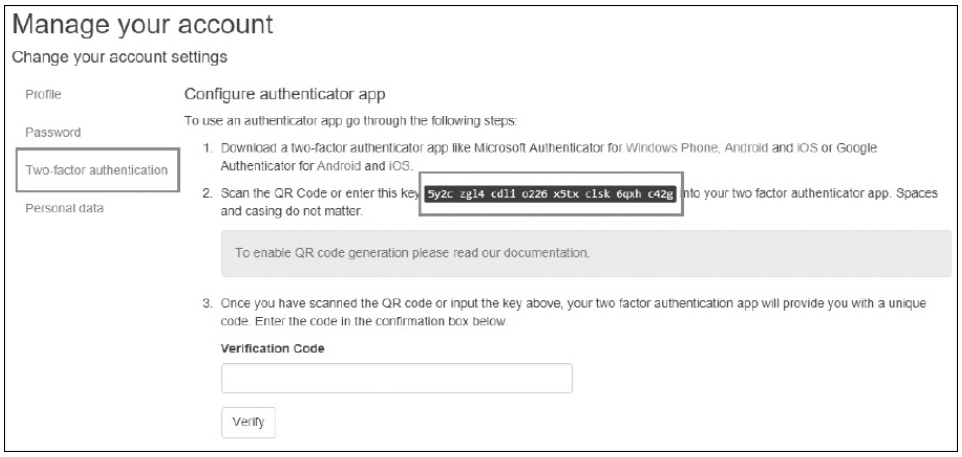


Figura 18.5 – Abilitazione della two-factor authentication tramite codice da inserire nell'authenticator.

La UI di ASP.NET Core Identity visualizza un token di 32 caratteri (e opzionalmente un QRCode) da usare con uno degli authenticator supportati, come Microsoft Authenticator e Google Authenticator, applicazioni per smartphone che è possibile ottenere dagli store di Windows, iOS e Android. La procedura si conclude digitando un codice di verifica fornito dall'autenticator. Una volta completata l'attivazione della two-factor authentication, i successivi login richiederanno sia l'immissione della password sia del codice a tempo fornito dall'authenticator, come è visibile nella Figura 18.6.

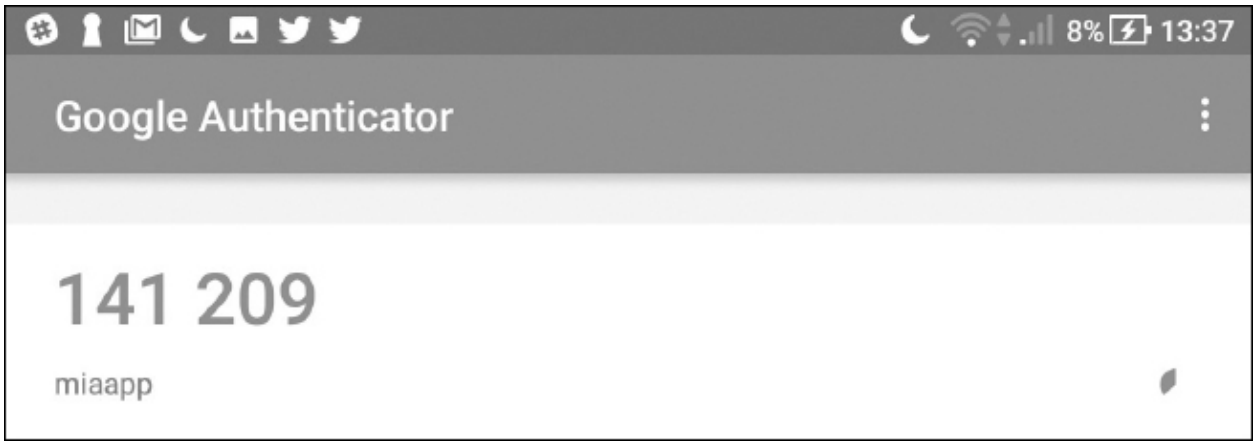


Figura 18.6 – L'authenticator fornisce un codice a tempo da inserire in fase di login.

Questa procedura presenta l'indubbio vantaggio di non richiedere l'invio di alcun messaggio SMS. Inoltre, la two-factor authentication è abilitata per default, quindi gli utenti possono iniziare a beneficiarne senza che sia richiesto alcun intervento da parte dello sviluppatore. Vediamo ora di quali altre tecniche dispone ASP.NET Core Identity per proteggere gli account degli utenti.

Protezione dai tentativi di accesso brute force

È risaputo che alcuni utenti scelgono password prevedibili per riuscire a ricordarle meglio. Come abbiamo visto in precedenza, possiamo imporre criteri per richiedere password più complesse ma questo può non essere sufficiente a scoraggiare un malintenzionato che vuole impadronirsi dell'account. Uno degli attacchi più facili e comuni consiste nel provare varie password nella speranza di indovinare quella corretta. ASP.NET Core Identity ci permette di proteggere gli account da questi tentativi di accesso brute force, imponendo un blocco dell'account.

Il `SignInManager` è il componente di ASP.NET Core Identity che si occupa di verificare le credenziali all'atto del login e, per default, registra anche il numero di tentativi consecutivi falliti. Al superamento di una certa soglia, l'account viene bloccato per un periodo configurabile, come nell'Esempio 18.13.

Esempio 18.13

```
services.AddDefaultIdentity<ApplicationUser>(options => {
   //Il lockout è abilitato per default
   //options.Lockout.AllowedForNewUsers = true;
   //Impostiamo durata del blocco e tentantivi massimi
   options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromHours(6);
   options.Lockout.MaxFailedAccessAttempts = 3;
})
.AddEntityFrameworkStores<ApplicationDbContext>();
```

In questo modo, il malintenzionato verrebbe bloccato per 6 ore al terzo login fallito. A ulteriori tentativi, la UI di ASP.NET Core Identity mostrerà il messaggio riportato nella Figura 18.7.

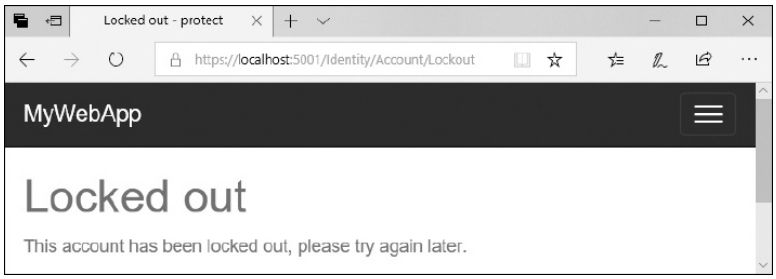


Figura 18.7 - L'account viene bloccato per precauzione dopo un certo numero di tentativi di login falliti.

Il lockout impedirà qualsiasi tentativo di accesso per il tempo configurato. Neanche il legittimo proprietario dell'account, in possesso della password corretta, riuscirebbe a fare il login in quel frangente. Questo meccanismo può quindi arrecare disagio all'utente a causa del comportamento di terzi ma è una misura efficace che impedisce conseguenze ben più gravi, come il furto dell'account. Dato che possiamo personalizzare la UI di ASP.NET Core Identity, potremmo aggiungere nella pagina le istruzioni per invitare l'utente a contattare l'assistenza tecnica affinché il blocco venga rimosso manualmente prima del termine.

La protezione dell'account passa anche dalle abitudini dell'utente stesso, che deve mantenere al sicuro le proprie credenziali e assicurarsi di fare il logout al termine di ogni sessione di utilizzo. Vediamo come possiamo porre rimedio con ASP.NET Core Identity quando questo non avviene.

Sign-out remoto

A volte capita che gli utenti dimentichino di fare il logout dopo aver usato una web application da un PC installato in un luogo pubblico, come in un internet café o nella hall di un albergo. Per impedire che persone estranee usino l'account, l'utente può effettuare il logout anche da remoto, semplicemente cambiando la password. Infatti, ogni cookie di autenticazione è emesso con un `SecurityStamp` che viene salvato nel database e rigenerato al cambio password. Ogni vecchio cookie di autenticazione perderà quindi la sua validità.

Verificare il `SecurityStamp` è un'operazione che richiede ad ASP.NET Core Identity di inviare una query al database e perciò, dato che comporta un seppur minimo costo prestazionale, viene fatto solo a intervalli regolari. Possiamo modificare tale intervallo come viene mostrato nell'Esempio 18.14.

Esempio 18.14

```
services.Configure<SecurityStampValidatorOptions>(options => {
    //Possiamo impostare TimeSpan.Zero per controllarlo ad ogni richiesta
    options.ValidationInterval = TimeSpan.FromMinutes(1);
});
```

Ora che abbiamo esaminato tutte le criticità in merito alla sicurezza degli account, vediamo come rendere più trasparente il modo in cui trattiamo i dati personali dei nostri utenti.

Conformità con le normative

A partire da ASP.NET Core 2.1, il team di sviluppo di Microsoft ha rivolto grande attenzione alle normative europee che regolano il trattamento dei dati personali e ha aggiunto funzionalità che agevolano gli sviluppatori nel loro adempimento. La normativa del GDPR, infatti, è un tema che interessa gli sviluppatori di tutto il mondo, dato che riguarda tutte le aziende che hanno clienti appartenenti alla Comunità Europea.

Come regola generale, dobbiamo fare in modo che l'utente sia informato sulle varie finalità di trattamento e possa concedere o negare il consenso per ogni finalità specifica. In mancanza del consenso, la nostra applicazione dovrà rinunciare a offrire il relativo servizio. Comunque, per assicurarci di essere conformi alla normativa, la migliore strategia è quella di affidarsi a un professionista che abbia competenze tecniche e legali specifiche.

Controllo dei dati da parte dell'utente per il GDPR

La normativa del GDPR prevede che l'utente sia sempre nel pieno controllo dei suoi dati e che quindi possa, in qualsiasi momento, consultare le informazioni in possesso dell'applicazione. Per agevolarci nell'adempimento di questo requisito, la UI di ASP.NET Core Identity mette a disposizione una pagina del profilo in cui l'utente può scaricare i propri dati in formato JSON ed eventualmente eliminare del tutto il suo account, come è illustrato nella Figura 18.8.

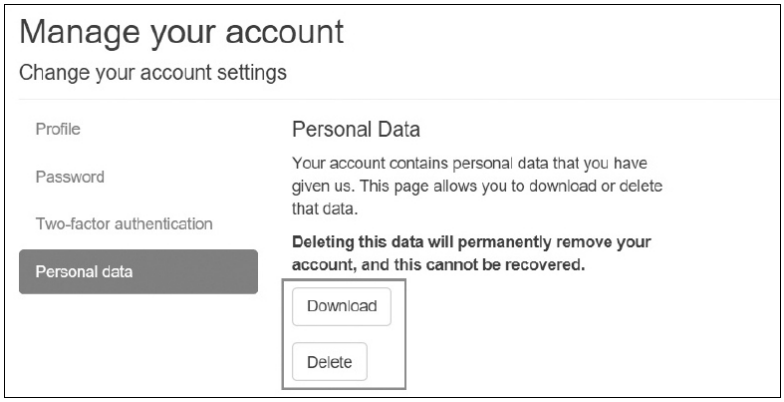


Figura 18.8 – Dalle pagine del profilo, l'utente può in qualsiasi momento scaricare o eliminare i propri dati.

Se abbiamo inserito nuove proprietà e relazioni nella classe `ApplicationUser`, dovremo fare lo scaffolding delle view `DownloadPersonalData.cshtml` e `DeletePersonalData.cshtml` per modificarne il comportamento, in modo che le operazioni di download ed eliminazione coinvolgano tutti i dati effettivamente presenti nel nostro database o sul disco fisso. Un esempio di personalizzazione lo troviamo nel blog post pubblicato da Microsoft all'indirizzo: http://aspit.co/bpb.

È importante tenere a mente che i dati dell'utente non possono essere usati indiscriminatamente per qualsiasi finalità, come, per esempio, quelle che riguardano il tracciamento delle sue abitudini per fini statistici. Di seguito vedremo come ASP.NET Core ci aiuta a mostrare un'informativa all'utente e a raccogliere il suo consenso.

Rispetto della normativa europea sui cookie

ASP.NET Core 2.1 rispetta anche la normativa europea sui cookie, che richiede all'applicazione di presentare un'informativa e di acquisire un consenso esplicito e specifico per ogni finalità di trattamento. A questo scopo, ogni template usa un `CookiePolicyMiddleware` che si occupa di presentare l'informativa in cima alla pagina, come è mostrato nella Figura 18.9.

Il contenuto dell'informativa può essere personalizzato agendo sulla partial view, che troviamo nel progetto al percorso `/Views/Shared/_CookieConsentPartial.cshtml`.

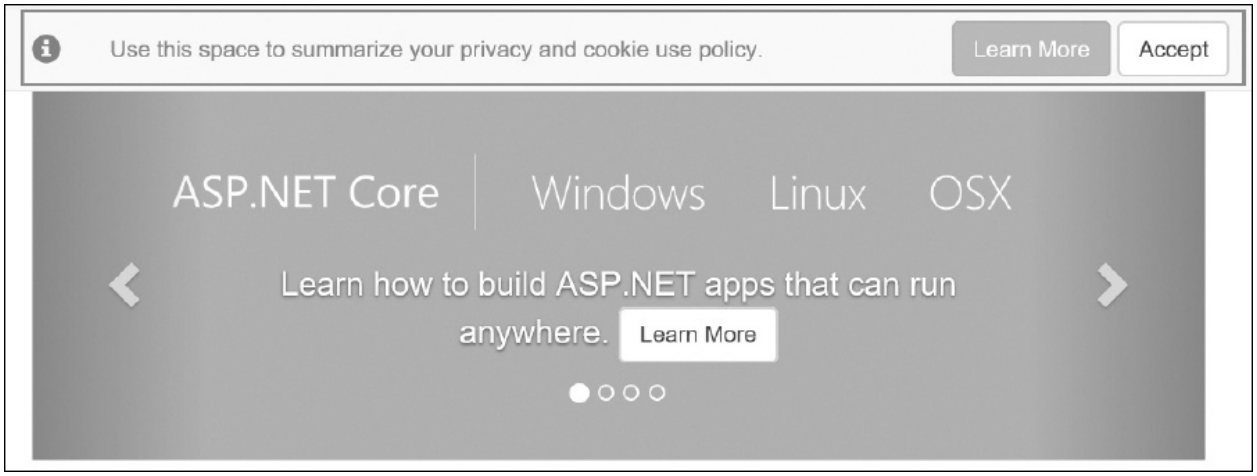


Figura 18.9 – UI per gli utenti che non hanno fornito il consenso all'emissione dei cookie non essenziali.

Nel caso in cui l'utente non abbia ancora fornito il consenso, il middleware ha anche lo scopo di impedire l'emissione dei cookie non essenziali, come, per esempio, quelli usati per la profilazione e il tracciamento delle abitudini di consumo.

I cookie di autenticazione sono esenti dal controllo, dato che sono considerati cookie essenziali. Secondo la legislazione, infatti, non occorre acquisire un esplicito consenso per emettere un cookie di autenticazione non persistente ma è sufficiente farne menzione all'interno di una pagina che riepiloga i termini di utilizzo dell'applicazione. L'utente potrà quindi fare il login anche senza fornire il consenso.

Ogni qualvolta emettiamo un qualsiasi altro tipo di cookie, abbiamo l'opportunità di decidere se considerarlo essenziale oppure no, agendo sulla proprietà IsEssential dell'oggetto di tipo `CookieOptions`, come mostrato nell'Esempio 18.15.

Esempio 18.15

```
var cookieOptions = new CookieOptions {
   Expires = DateTimeOffset.Now.AddMonths(1),
   IsEssential = false
};
HttpContext.Response.Cookies.Append("TrackId", "128432", cookieOptions);
```

Così, in maniera trasparente, possiamo emettere un cookie non essenziale nella risposta HTTP solo se l'utente aveva esplicitamente prestato il suo consenso.

Con questo si conclude la nostra esplorazione della UI di ASP.NET Core Identity. Passiamo ora a vedere come usare la sua API lato server per compiere operazioni sul database di utenti locali.

Gestire gli utenti con ASP.NET Core Identity

Oltre alla UI che abbiamo appena visto, ASP.NET Core Identity espone una ricca API per gestire gli utenti locali in maniera programmatica. Il punto di ingresso principale in questa API è rappresentato dalla classe `UserManager<TUser>` di cui possiamo far uso da pagine riservate agli amministratori del sito, in modo da permetterle di compiere svariate attività:

- Creare, modificare, bloccare ed eliminare gli utenti locali.
- Assegnare i claim, ruoli compresi, che popoleranno la `ClaimsIdentity` dopo il login.
- Reimpostare la password o confermare l'indirizzo e-mail dell'utente, nel caso in cui abbia richiesto assistenza tecnica per compiere queste operazioni.
- Interrogare il database locale, per estrarre uno o più utenti rispondenti a determinati criteri.

Lo `UserManager<TUser>` è un servizio registrato da ASP.NET Core Identity e perciò lo possiamo ricevere facilmente come dipendenza nei controller, come dimostra l'Esempio 18.16.

Esempio 18.16

```
public class UserManagementController : Controller
{
   private readonly UserManager<ApplicationUser> userManager;
   public UserManagementController(UserManager<ApplicationUser> userManager)
   {
      this.userManager = userManager;
   }
   //Qui le action che usereanno lo userManager per compiere operazioni
}
```

Il type parameter `TUser` rappresenta l'entità dell'utente, per esempio, `ApplicationUser`.

Sfruttando lo `UserManager` abbiamo la garanzia che ogni operazione, come, per esempio, la reimpostazione di una password, venga eseguita in sicurezza. Salvo rare eccezioni, non dovremmo usare comandi SQL per aggiornare direttamente il database degli utenti locali ma dovremmo sempre sfruttare lo `UserManager` per qualsiasi tipo di operazione, a partire dalla creazione.

Creare un utente programmaticamente

Quando realizziamo un'applicazione destinata a un numero limitato di utenti noti, può essere conveniente creare a priori gli account e poi comunicare loro le credenziali per invitarli ad accedere. In situazioni come questa, possiamo preparare tutti gli account con il metodo `CreateAsync` dell'oggetto `UserManager<TUser>`. L'Esempio 18.17 mostra la creazione di un nuovo account fornendo l'e-mail, lo username e la password di l'accesso.

Esempio 18.17

```
var user = new ApplicationUser {
   Email = "user1@example.com",
   UserName = "user1@example.com",
   EmailConfirmed = true
};
await userManager.CreateAsync(user, "Password1!");
```

In questa fase abbiamo l'opportunità di valorizzare anche le proprietà personalizzate della classe `ApplicationUser`, comprese le eventuali entità correlate. La password verrà elaborata da ASP.NET Core Identity e memorizzata in maniera sicura, come già abbiamo visto. Subito dopo aver creato gli account, possiamo ovviamente recuperarli dal database locale, per verificare che siano stati creati correttamente.

Interrogare il database degli utenti locali

Dato che ASP.NET Core Identity, dietro le quinte, può usare un O/RM come Entity Framework Core, offre tutta l'espressività delle query LINQ per filtrare, ordinare e paginare gli utenti memorizzati nel database locale. Nell'Esempio 18.18 adoperiamo la proprietà `Users`, di tipo `IQueryable<TUser>`, per ottenere un sottoinsieme di utenti.

Esempio 18.18

```
var users = await userManager.Users
   .Where(user => user.Email.EndsWith("@example.com"))
   .OrderBy(user => user.Email)
.ToListAsync();
```

Se conosciamo esattamente lo username, l'e-mail o l'identificativo dell'utente che intendiamo recuperare, lo `UserManager` espone dei metodi specializzati per ottenere un singolo utente, ovvero `FindByNameAsync`, `FindByEmailAsync` e `FindByIdAsync`. Ottenere un account in questo modo è il primo passo da compiere quando vogliamo apportare modifiche.

Assegnare claim e modificare i dati di login dell'utente

Oltre alle informazioni di login come e-mail e password, un utente è descritto dai suoi claim che, come abbiamo già visto nel capitolo precedente, possiamo impiegare nelle policy di autorizzazione, per determinare se può accedere o no a determinate funzionalità dell'applicazione. Per gestire i claim di un utente esistente, lo `UserManager` offre il metodo `AddClaimAsync`, come è illustrato nell'Esempio 18.19. Poi, usiamo anche il metodo `UpdateAsync` per aggiornare i dati di login.

Esempio 18.19

```
//Estraggo l'utente in base alla sua e-mail
var user = await userManager.FindByEmailAsync("user1@example.com");

//Creo un claim e glielo associo
var claim = new Claim(ClaimTypes.Role, "Customer");
await userManager.AddClaimAsync(user, claim);

//Aggiorno la password
user.PasswordHash = userManager.PasswordHasher.HashPassword(user, "New1!");
await userManager.UpdateAsync(user);
```

Allo stesso modo, i claim possono essere rimossi usando il metodo `RemoveClaimAsync`. Lo `UserManager` dispone inoltre di molti metodi specializzati nell'aggiornamento degli utenti e, alcuni di essi, sono dimostrati dall'applicazione `user-administration` che si trova allegata al capitolo.

In questa prima parte del capitolo li abbiamo strettamente legati alla gestione degli account, mentre ora passiamo a vedere quali sono gli altri accorgimenti di sicurezza adottati da ASP.NET Core.

Proteggersi dagli attacchi

Parte della sicurezza delle applicazioni è delegata alla robustezza del web server e del sistema operativo, che impediscono ai malintenzionati di introdursi nel server sfruttando delle falle. Questi rischi si possono ridurre a livello sistemistico tenendo il sistema operativo sempre aggiornato e usando password robuste, IP filtering e logging per i servizi di gestione come FTP, SSH o Remote Desktop.

Anche gli sviluppatori sono responsabili della sicurezza dell'intero sistema ed è importante imparare a riconoscere i vari tipi di attacchi che i malintenzionati possono apportare a livello applicativo per introdursi nella macchina o per rubare gli account esistenti.

Proteggere l'applicazione con un certificato SSL

Quando gli utenti visitano l'applicazione su protocollo HTTP, rischiano che i propri dati, comprese le password di accesso, vengano intercettate perché viaggiano in chiaro su rete internet. Il rischio è concreto su reti WiFi pubbliche, come quelle di aeroporti, hotel ed esercizi commerciali, in cui è più probabile che vi siano i cosiddetti "man-in-the-middle" a intercettare il traffico tra client e server.

Per evitare questo problema, possiamo procurarci un certificato SSL da una certification authority riconosciuta e forzare l'uso del protocollo HTTPS. In questo modo, ogni informazione viene cifrata e viaggia in maniera confidenziale su una connessione sicura. Anche nel caso di un attacco man-in-the-middle, l'utente ne avrebbe evidenza perché il browser riporterebbe il sito come "Non sicuro" e gli consiglierebbe di non proseguire nella sua navigazione.

Procurarsi un certificato SSL valido e gratuito è semplice e automatico usando una certification authority come Let's Encrypt. Questo è un importante incentivo per usare il protocollo HTTPS in ogni genere di applicazione. L'obiettivo non è solo quello di proteggere i dati personali ma anche i testi cercati, le richieste di contatto e qualsiasi informazione digitata in un form, che possono rivelare le preferenze dell'utente.

Oltretutto, come vediamo nella Figura 18.10, da luglio 2018 Google Chrome segnala come "Non sicuri" i siti sprovvisti di certificato e questo potrebbe già essere un motivo sufficiente a disincentivare l'utente nell'usare la nostra applicazione.

A partire da ASP.NET Core 2.1, tutte le nuove applicazioni sono dotate di un certificato autofirmato, in modo che possiamo sperimentare l'uso di HTTPS sin dalle prime fase di sviluppo. Così, Microsoft vuole sensibilizzarci sull'importanza di usare un certificato su ogni nostra applicazione.

Prima di entrare in produzione, dovremo procurarci un certificato SSL emesso da una certification authority riconosciuta e la procedura di installazione varierà in base alla piattaforma che abbiamo scelto di usare. Per esempio, se la nostra applicazione ASP.NET Core è ospitata da Kestrel alle spalle di un reverse proxy, allora il certificato andrà installato in IIS, NGINX o Apache. Approfondiremo questo argomento nel corso del Capitolo 22.

Figura 18.10 – Google Chrome indica come "Non sicuro" un sito sprovvisto di certificato SSL.

Mantenere il certificato anche su Kestrel è comunque una buona pratica perché ci permette di proteggere anche il traffico che arriva dal reverse proxy. Un accorgimento come questo ci aiuta a realizzare la defense-in-depth, ovvero un irrobustimento generale del sistema che deriva dall'apportare misure di sicurezza su più livelli. L'Esempio 18.20 ci mostra come preparare il web host dalla classe `Program`, in modo da usare uno o più certificati SSL con Kestrel.

Esempio 18.20

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
   .UseKestrel((context, options) => {
   options.Listen(IPAddress.Loopback, 5000, listenOptions => {
      listenOptions.UseHttps(httpsOptions =>
      {
         //Possiamo leggere il certificato da file
         var cert = new X509Certificate2(@"cert.pfx", "password");
         //Oppure dal certificate store (su Windows)
         //var cert = CertificateLoader.LoadFromStoreCert(
            "localhost", "My", StoreLocation.CurrentUser, allowInvalid: false);

         //Impostiamo un singolo certificato
         httpsOptions.ServerCertificate = localhostCert;
         //Oppure usiamo SNI per selezionare il certificato in base all'host
         //httpsOptions.ServerCertificateSelector = (context, nomeHost) => {
         // return localhostCert;
         //};
      });
   });
})
.UseStartup<Startup>();
```

Se la nostra è un'applicazione multi-tenant, è probabile che dovremo renderla accessibile da molti nomi host diversi, uno per ogni tenant. Sia Kestrel sia HTTP.sys hanno il supporto a SNI (Server Name Indication), che ci consente appunto di usare svariati certificati SSL sullo stesso indirizzo IP. L'Esempio 18.21 mostra invece come configurare HTTPS con il web server HTTP.sys.

Esempio 18.21

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
   .UseHttpSys(options => {
      options.UrlPrefixes.Add("https://www.example.com:443");
      options.UrlPrefixes.Add("https://tenant1.example.com:443");
      //...
   })
.UseStartup<Startup>();
```

HTTP.sys richiede che i certificati siano configurati su Windows con il comando `netsh http add sslcert`, come indicato all'indirizzo: `http://aspit.co/bol`.

In generale, se vogliamo approfondire le opzioni di installazione e configurazione del certificato, Microsoft ha pubblicato un blog post da leggere: `http://aspit.co/bok`.

A questo punto, vediamo come assicurarci che le richieste degli utenti arrivino su HTTPS.

Forzare il traffico su HTTPS

Dal momento che la nostra applicazione sfrutta un certificato SSL, dobbiamo fare il possibile per portare i nostri utenti a inviare richieste su HTTPS, in modo da proteggerli durante tutte le fasi della sessione di navigazione. Un primo intervento che possiamo adottare consiste nel decorare i controller e le action con l'attributo `[RequireHttps]`, che si può configurare anche come filtro globale dal metodo `ConfigureServices` della classe `Startup`, come nell'Esempio 18.22.

Esempio 18.22

```
services.AddMvc(options => {
   options.Filters.Add(new RequireHttpsAttribute());
   //Decommentiamo questa riga se vogliamo indicare una porta diversa
   //options.SslPort = 443;
});
```

In questo modo, ogni richiesta inviata su HTTP a un'action di ASP.NET Core MVC verrà subito reindirizzata verso HTTPS. Tuttavia, possiamo essere ancora più efficaci di così, perché questa soluzione non copre le richieste che esulano da ASP.NET Core MVC, come quelle inviate ai file statici. L'Esempio 18.23 mostra una soluzione che sfrutta l'apposito `HttpsRedirectionMiddleware`, che coprirà ogni richiesta HTTP. Il codice è inserito nel metodo `Configure` della classe `Startup`.

Esempio 18.23

```
app.UseHttpsRedirection();
```

Inoltre, è possibile informare il browser che la nostra applicazione deve sempre essere chiamata su HTTPS, anche se l'utente dovesse esplicitamente digitare `http://` nella barra degli indirizzi. Il meccanismo dell'HSTS (HTTP Strict Transport Security) serve a ridurre drasticamente il numero di richieste su connessione insicura e si basa su una semplice intestazione inclusa nella risposta. Nell'Esempio 18.24 usiamo il middleware che si occupa di aggiungere tale intestazione, che poi verrà interpretata e messa in cache dal browser.

Esempio 18.24

```
//Produrrà l'intestazione che indica al browser di fare richieste su HTTPS
//Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
app.UseHsts();
```

Il middleware ammette delle opzioni di configurazione per regolare la durata in cache o l'inclusione dei sottodomini. Per controllare questi aspetti, usiamo anche l'extension method `AddHsts` dal metodo `ConfigureServices` della classe `Startup`.

Teniamo presente che queste soluzioni sono efficaci nelle web application ASP.NET Core MVC. Se stiamo invece realizzando una Web API, la soluzione migliore consiste nel fornire subito un errore 400 (Bad Request), per dissuadere il client dal continuare a inviare informazioni in chiaro su HTTP. Ora che abbiamo messo in sicurezza la connessione con l'utente, dobbiamo imparare a riconoscere i vari tipi di attacco e proteggere l'applicazione da essi.

Cross Site Scripting (XSS)

Uno degli attacchi più noti è il Cross Site Scripting, che consiste nell'inserire codice JavaScript o markup malevolo nelle caselle di testo adibite all'invio di commenti o altro tipo di contenuti da parte degli utenti. Dato che tali contenuti vengono poi visualizzati pubblicamente nelle pagine del sito, potrebbero causare danni agli altri utenti, come il furto dei cookie di autenticazione o reindirizzamenti indesiderati verso siti malevoli.

La soluzione ideale consiste nel non rappresentare mai il contenuto inviato dagli utenti così com'è, ma sanitizzarlo con l'HTML encoding, per rendere gli script e il markup inoffensivi. Finché usiamo le view Razor di ASP.NET Core MVC non dobbiamo preoccuparci di nulla, perché questo compito è svolto automaticamente da un `HtmlEncoder`. Ipotizziamo di aver realizzato una view Razor come quella dell'Esempio 18.25.

Esempio 18.25

```
@{
   //In un'applicazione reale questo valore arriva dal database
   var contenuto = "<iframe src='http://www.sitomalevolo.com'></iframe>";
}
<h4>L'utente ha scritto:</h4> <p>@contenuto</p>
```

Senza aver usato alcun accorgimento particolare, il codice HTML, che causerebbe l'inserimento di un `iframe` nella pagina, viene invece reso inoffensivo trasformando i caratteri < e > nelle loro controparti html `<` e `>` e, come risultato, apparirà in forma letterale, come nella Figura 18.11.

L'utente ha scritto:

<iframe src='http://www.sitomalevolo.com'></iframe>

Figura 18.11 – Il contenuto malevolo inviato da un utente viene reso inoffensivo dall'HtmlEncoder.

In alcuni casi, questo comportamento potrebbe risultare indesiderato, per esempio, quando dobbiamo rappresentare un frammento di HTML prodotto da noi stessi e che quindi sappiamo essere sicuro. In questa situazione, possiamo usare l'HTML helper `@Html.Raw` per evitare che l'`HtmlEncoder` operi delle trasformazioni, come mostrato nell'Esempio 18.26.

Esempio 18.26

```
@Html.Raw("<strong>Questo testo è rappresentato in grassetto</strong>")
```

Inoltre, la funzionalità del servizio `HtmlEncoder` può anche essere usata on-demand, sfruttando la dependency injection. ASP.NET Core fornisce, in modo simile, anche i servizi `JavaScriptEncoder` e `UrlEncoder` per trasformare i contenuti e renderli sicuri da utilizzare con JavaScript o da passare via URL. Tutti gli encoder possiedono il metodo `Encode` predisposto a tale scopo.

Cross Site Request Forgery (CSRF o XSRF)

Nel capitolo precedente abbiamo visto come, in seguito al login, ASP.NET Core Identity emetta un cookie di autenticazione che permette all'utente di essere riconosciuto in tutte le successive richieste. Questo fatto può essere sfruttato da un eventuale attaccante per compiere operazioni usando le autorizzazioni concesse all'utente ma senza che egli ne sia consapevole. L'attacco può iniziare con una e-mail di phishing, che attira l'utente nel sito del malintenzionato con la promessa di un premio. L'utente, cliccando il link contenuto nell'e-mail, atterrerà in una pagina contenente un form come quello illustrato dall'Esempio 18.27.

Esempio 18.27

```
<form action="http://example.com/account" method="post">
   <input type="hidden" name="Importo" value="10000">
   <input type="hidden" name="Iban" value="IT80R111111111111111111">
   <input type="submit" name="InviaBonifico" value="Ricevi il premio!">
</form>
```

Come si può notare dall'esempio, nel form sono presenti dei campi hidden creati ad arte e che verranno interpretati da example.com, l'applicazione attaccata. Quando l'utente clicca il bottone, non si rende conto che anziché ricevere un premio sta in realtà inviando una richiesta POST verso example. com e, se era loggato, il browser includerà anche il cookie di autenticazione. Questo permette all'attaccante di far compiere un'operazione autenticata all'utente senza entrare mai in possesso del suo cookie o delle sue credenziali.

Per evitare questo genere di abusi, le view Razor di un'applicazione ASP.NET Core MVC includono un campo hidden `RequestVerificationToken` ogni volta che viene usato il tag helper `<form>`. La Figura 18.12 illustra l'output HTML che viene prodotto dalla view.

```
◢ <form method="post">
    <input type="submit" value="Invia" />
    <input name="__RequestVerificationToken" type="hidden" value="CfDJ8HYl1dua..." />
  </form>
```

Figura 18.12 – Creando un form in una view, viene automaticamente aggiunto il token anti-XSFR.

Dato che il token è un codice casuale che l'attaccante non può assolutamente indovinare, le richieste inviate dal form malevolo non potranno avere successo. Infatti, a fronte di un token mancante o incorretto, ASP.NET Core MVC risponderà con un errore. Tutto quello che bisogna fare per sfruttare questo meccanismo è porre l'attributo `[AutoValidateAntiforgeryToken]` sulle action interessate, oppure configurarlo come filtro globale, come è mostrato nell'Esempio 18.28.

Esempio 18.28

```
services.AddMvc(options => {
   options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute());
});
```

Ogni richiesta inviata con i metodi `POST`, `PUT` o `DELETE` verrà sottoposta a validazione, mentre le richieste di tipo `GET` non sono soggette a questo tipo di attacco, dato che tipicamente vengono usate per la lettura di dati e non per la scrittura. Vediamo ora un altro tipo di attacco mirato al furto dell'account.

Open Redirect

L'attacco Open Redirect è uno dei meno noti ma anche uno dei più subdoli. L'attacco ha lo scopo di rubare le credenziali e inizia tipicamente con una e-mail di phishing allarmante che invita l'utente a cliccare un link per fare subito il login. Il dominio dell'url sembra essere quello corretto (per esempio, example.com) ma la chiave querystring `returnUrl` ne contiene uno diverso, come si può vedere nell'Esempio 18.29.

Esempio 18.29

```
https://example.com/login?returnUrl=https%3A%2F%2www.exxxample.com
```

L'attacco sfrutta il fatto che alcune applicazioni, a login avvenuto con successo, reindirizzano l'utente alla pagina di provenienza, indicata in querystring con il nome `returnUrl`. L'utente verrà quindi reindirizzato al sito malevolo e gli verrà presentata una pagina di login contraffatta, in tutto e per tutto simile a quella originale, che tenta di fargli credere che il login non sia andato a buon fine.

L'utente, attirato dal messaggio d'errore, potrebbe non accorgersi che si trova su un altro dominio e, reinserendo le credenziali, di fatto le consegnerà nelle mani del malintenzionato.

Se usiamo la UI di ASP.NET Core Identity, siamo già protetti da questo genere di attacco mentre, se abbiamo scelto una procedura di login personalizzata, possiamo proteggerci usando il metodo `LocalRedirect` nella nostra action di login, come è mostrato nell'Esempio 18.30.

Esempio 18.30

```
[HttpPost]
public async Task<IActionResult> Login(LoginModel login, string returnUrl)
{
   if (ModelState.IsValid)
   {
      //Procedura di verifica delle credenziali personalizzata
      if (await Verifica(loginModel.Username, loginModel.Password))
      {
         //Usiamo LocalRedirect per evitare gli attacchi Open Redirect
         return LocalRedirect(returnUrl);
      }
   }
   return View(loginViewModel);
}
```

Il metodo `LocalRedirect` solleverà un'eccezione nel caso in cui il `returnUrl` non sia locale. In alternativa, possiamo usare il metodo `Url.IsLocalUrl` per ottenere un valore booleano e, nel caso risultasse `false`, compiere altre operazioni, come loggare il tentativo di attacco.

Spostiamoci ora lato client, per esaminare un aspetto legato all'abuso di Web API da parte di terzi. Non si tratta propriamente di un attacco ma di un comportamento tenuto dai browser quando le pagine HTML inviano richieste AJAX a differenti origini.

Cross-Origin Resource Sharing (CORS)

Di norma, i browser consentono al codice lato client di inviare richieste AJAX solo verso URL che condividono la sua stessa origine, cioè aventi lo stesso nome host, la stessa porta e lo stesso protocollo. Perciò, se il codice client-side della web application fosse pubblicato in un altro dominio, vedremmo la richiesta AJAX fallire con un errore in console, come nella Figura 18.13.

Figura 18.13 – Per default, una richiesta AJAX fallisce se viene inviata a un URL con un'altra origine.

Questo è un meccanismo di protezione che serve a evitare che applicazioni client di terze parti usino indiscriminatamente le nostre WebAPI.

Se entrambe le origini ci appartengono, Il blocco può essere evitato abilitando Cross-Origin Resource Sharing (CORS) nella nostra WebAPI. Il browser, infatti, non blocca subito le richieste cross-origin ma interpella il server della WebAPI con una cosiddetta richiesta pre-flight, per capire se ha disposizioni in merito. L'Esempio 18.31 mostra come usare l'extension method `AddCors` nel metodo `ConfigureServices` della classe `Startup`, per definire una policy CORS che abilita in maniera selettiva alcune origini client.

Esempio 18.31

```
services.AddCors(options => {
   options.AddPolicy("AllowOrigins", builder => {
      //Consentiamo richieste con qualsiasi intestazione e metodo http
      builder.AllowAnyHeader();
      builder.AllowAnyMethod();
      //Abilitiamo queste due origini a chiamare api.example.com
      builder.WithOrigins("http://www.example.com", "http://example.com");
      //Se invece vogliamo abilitare ogni origine, usiamo:
      //builder.AllowAnyOrigin();
   });
});
```

Ora, come mostrato nell'Esempio 18.32, usiamo gli attributi `[EnableCors]` e `[DisableCors]` in corrispondenza dei controller o delle action su cui rendere effettiva la policy CORS appena definita.

Esempio 18.32

```
[EnableCors(policyName: "AllowOrigins"), Route("api/[controller]")]
public class CustomerController : ControllerBase {
   // Le action definite qui useranno la policy Cors "AllowOrigins"
   // Escludiamo questa action con l'attributo DisableCors
   [HttpPost, DisableCors]
   public void Post([FromBody] string value) {
   }
}
```

In alternativa, la policy CORS può essere applicata a livello globale, in modo che abbia effetto per ogni controller. In questo caso, usiamo l'apposito `CorsMiddleware`, come si può vedere nell'Esempio 18.33.

Esempio 18.33

```
app.UseCors("AllowOrigins");
```

Così, la richiesta AJAX potrà avere successo perché l'origine è stata esplicitamente indicata.

È importante considerare che non possiamo fare affidamento solo su CORS per autorizzare l'accesso alla WebAPI. Anche se i browser impediscono le richieste cross-origin, infatti, nulla vieta a sviluppatori di terze parti di accedere alla nostra WebAPI con codice lato server o con applicazioni desktop e mobile native. Per questo motivo è sempre importante proteggere la nostra WebAPI con l'autenticazione, come discusso nel corso del capitolo precedente.

Ora che abbiamo messo in sicurezza la parte client dell'applicazione, iniziamo a vedere come proteggere i dati lato server, a partire dalla configurazione dell'applicazione che può contenere essa stessa dei dati sensibili, come API Key e stringhe di connessione.

Configurare l'applicazione in maniera sicura

Nel corso del Capitolo 3 abbiamo visto come configurare l'applicazione ASP.NET Core usando varie fonti, tra cui i file di testo come `appsettings.json`. Tipicamente, i file di testo sono sottoposti a controllo di versione insieme al codice sorgente dell'applicazione e perciò non sono consigliati per contenere valori di configurazione sensibili, come API key o chiavi di crittografia, che andrebbero tenuti segreti. Se li aggiungessimo al controllo di versione, infatti, verrebbero divulgati a tutti coloro che hanno accesso al repository, che non necessariamente devono conoscere tali valori. Il problema è particolarmente sentito nei progetti open-source, dove il codice è pubblicamente accessibile da chiunque. Inoltre, anche le stringhe di connessione contengono informazioni segrete come username e password e, dato che cambiano in base all'ambiente di deploy, è opportuno valutare altre fonti di configurazione, che mantengano i valori in uno spazio di archiviazione separato dal progetto.

- Nell'ambiente di sviluppo possiamo valutare user secrets o variabili d'ambiente.
- Nell'ambiente di staging e produzione Azure Key Vault o variabili d'ambiente.

Nel corso del Capitolo 3 abbiamo già visto come usare le variabili d'ambiente, quindi ci concentreremo su user secrets e Azure Key Vault.

User secrets

Gli user secrets sono una fonte alternativa alle variabili d'ambiente da usare in ambiente di sviluppo. I valori di configurazione che gli affidiamo sono memorizzati all'interno della home directory del nostro utente, perciò in uno spazio separato da quello in cui si trova il progetto. I valori sono salvati in chiaro ma questo è raramente un problema dato che, anche in caso di furto della nostra macchina di sviluppo, il contenuto della home directory è tipicamente accessibile solo dopo aver fatto il login con il nostro utente.

Prima di iniziare a usare user secrets, verifichiamo che all'interno del file .csproj ci sia un elemento `UserSecretsId` come si può vedere nell'Esempio 18.34. Creando un'applicazione che include ASP.NET Core Identity, tale elemento sarà già presente e gli utenti di Visual Studio possono generarlo automaticamente cliccando il progetto con il tasto destro e selezionando Manage user secrets dal menu contestuale. È importante che il suo valore sia univoco per ogni progetto, dato che viene usato come chiave per mantenere isolati i vari contenitori.

Esempio 18.34

```
<PropertyGroup>
   <TargetFramework>netcoreapp2.1</TargetFramework>
   <UserSecretsId>aspnet-myapp-B3998871-F750…</UserSecretsId>
</PropertyGroup>
```

La fonte degli user secrets è già presente in ogni applicazione ASP.NET Core in cui sia stato usato il web host builder di default, quindi non sarà necessario apportare ulteriori modifiche al codice. Ci basta usare il sottocomando dotnet user-secrets per impostare le nostre chiavi di configurazione, come nell'Esempio 18.35. Tali valori saranno poi accessibili dall'applicazione nel modo consueto, per esempio, usando la proprietà `Configuration`, che viene esposta dalla classe `Startup` o il servizio `IOptionsMonitor<T>` già visto nel Capitolo 3.

Esempio 18.35

```
dotnet user-secrets set Smtp:Host smtp.example1.com
```

I due punti sono usati come carattere separatore nel percorso di una chiave gerarchica. I valori invece sono salvati nel file `secrets.json` situato in una sottodirectory della home dell'utente. Lo specifico percorso è riepilogato dalla Tabella 18.1 per ogni sistema operativo.

Tabella 18.1 – Percorsi di archiviazione degli User secrets nei vari sistemi operativi.

Sistema operativo	Percorso di archiviazione
Windows	`%APPDATA%\microsoft\UserSecrets\`*`UserSecretsId`*`\`
Linux	`~/.microsoft/usersecrets/`*`UserSecretsId`*`/`
macOS	`~/.microsoft/usersecrets/`*`UserSecretsId`*`/`

Il file `secrets.json` può essere modificato manualmente e gli utenti di Visual Studio possono raggiungerlo velocemente con la funzionalità Manage user secrets già citata. Affinché le nuove modifiche abbiano effetto, l'applicazione ASP.NET Core dovrà essere riavviata.

Essendo una fonte aggiunta in maniera prioritaria rispetto ai file di testo, user secrets è anche una buona soluzione per ridefinire facilmente i valori presenti in `appsettings.json`, così che ogni sviluppatore possa adattarli al suo ambiente di sviluppo senza apportare modifiche al file.

Azure Key Vault

Per l'ambiente di produzione, possiamo valutare Azure Key Vault, un servizio cloud per l'archiviazione sicura di chiavi crittografiche e ogni altro genere di valore di configurazione da tenere segreto. I valori sono protetti da crittografia at-rest, eventualmente supportata da moduli hardware dedicati (HMS), in modo da garantire un insuperabile livello di protezione. Inoltre, Azure Key Vault attua anche la crittografia in-transit, in modo che i valori di configurazione possano essere usati in maniera sicura anche all'esterno del data center di Azure, cioè da applicazioni on-premise. L'accesso alle chiavi è regolato in lettura e in scrittura da un dettagliato sistema di permessi retto da Azure Active Directory, che consente di autorizzare utenti, applicazioni o macchine virtuali.

Iniziamo dal pacchetto NuGet `Microsoft.Extensions.Configuration.AzureKeyVault`, che deve essere installato nel progetto, e poi modifichiamo la classe `Program` per aggiungere Azure Key Vault come ulteriore fonte di configurazione, come viene mostrato nell'Esempio 18.36.

Esempio 18.36

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
   .ConfigureAppConfiguration((ctx, builder) => {
      var conf = builder.Build();
      var endpoint = conf.GetValue<string>("VaultEndpoint");
      var provider = new AzureServiceTokenProvider();
      var callback = provider.KeyVaultTokenCallback;
      var authCallback = new KeyVaultClient.AuthenticationCallback(callback);
      var keyVaultClient = new KeyVaultClient(authCallback);
      var secretManager = new DefaultKeyVaultSecretManager();
      builder.AddAzureKeyVault(endpoint, keyVaultClient, secretManager);
```

```
}).UseStartup<Startup>();
```

Per la corretta configurazione del servizio, abbiamo bisogno di recuperare alcune informazioni dal portale di Azure: sono riepilogate nella Tabella 18.2. Per ottenerle, dobbiamo per prima cosa registrare l'applicazione in Azure Active Directory, seguendo gli stessi passi che abbiamo già compiuto nel capitolo precedente, parlando di autenticazione con Azure Active Directory. Ovviamente, queste informazioni devono restare confidenziali e perciò è necessario che vengano memorizzate su variabili d'ambiente anziché nel file `appsettings.json`.

Tabella 18.2 – Variabili d'ambiente da impostare per usare Azure Key Vault.

Variable d'ambiente	Come ottenere il valore
ClientId	ID della registrazione dell'applicazione web in Azure Active Directory
ClientSecret	Chiave creata dalla blade della registrazione, in "Settings" > "Keys"
VaultEndpoint	L'URL di Azure Key Vault, riportato nella sua blade

All'avvio dell'applicazione, verranno ottenute da Azure Key Vault tutte le chiavi di configurazione che avremo definito dal portale di Azure. Il sistema è molto versatile perché ci consente di aggiungere, modificare e definire le date di scadenza delle chiavi da interfaccia web, come si può vedere nella Figura 18.14.

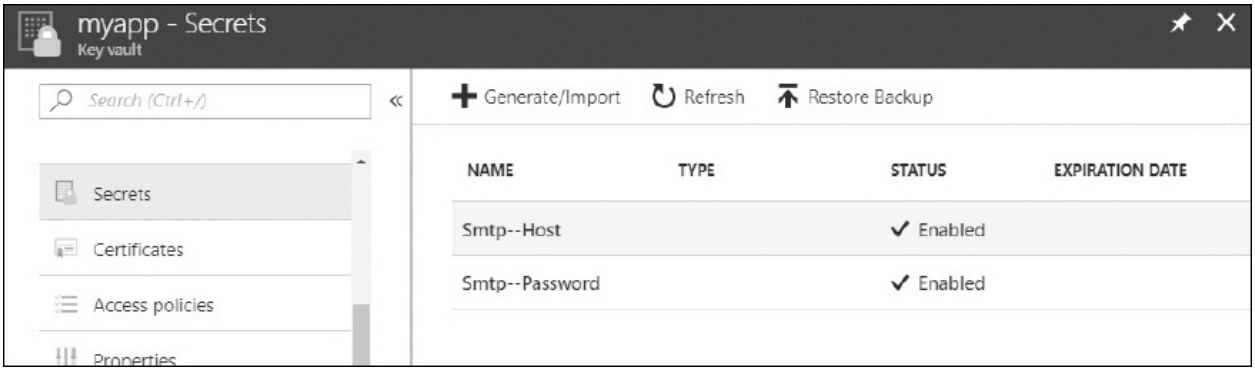


Figura 18.14 – I valori segreti si gestiscono in Azure Key Vault, con un'interfaccia web dal portale di Azure.

In questo modo abbiamo un punto centralizzato in cui gestire i valori di configurazione, che eventualmente possono essere sfruttati da più applicazioni contemporaneamente. Dall'applicazione ASP.NET Core, usiamo l'approccio consueto per accedere ai valori: impieghiamo la proprietà `Configuration` esposta dalla classe `Startup` oppure il servizio `IOptionsMonitor<T>`.

Ora che la configurazione è protetta, vediamo anche come proteggere anche gli altri tipi di dato, come, per esempio, i dati personali dei nostri utenti.

Proteggere i dati sensibili

ASP.NET Core usa la cifratura dei dati in molte situazioni, come, per esempio, durante la generazione dell'AntiForgeryToken o dei cookie di autenticazione. Cifrare e verificare l'integrità dei dati è essenziale per vanificare i tentativi di manipolazione. È per questo motivo che ASP.NET Core impiega una Data Protection API che anche noi possiamo sfruttare per attuare una cifratura at-rest di dati arbitrari. La cifratura avviene con l'algoritmo simmetrico AES e validazione SHA256, così da proteggere i dati sensibili più efficacemente quando li memorizziamo sul disco o nel database. Questa pratica aggiunge un ulteriore livello di protezione contro il furto di dati.

Usare Data Protection API per cifrare dati

La Data Protection API è già pre-configurata in ogni applicazione ASP.NET Core e tutto quel che dobbiamo fare è usare il servizio `IDataProtectionProvider`, come si può notare nell'Esempio 18.37, che illustra un'action di ASP.NET Core MVC, il cui scopo è salvare un codice fiscale in maniera protetta.

Esempio 18.37

```
[HttpPost]
public IActionResult UpdateFiscalCode(string userId, string fiscalCode,
                      [FromServices] IDataProtectionProvider provider)
{
   //Otteniamo un protector per il purpose "userData"
   IDataProtector protector = provider.CreateProtector("userData");
```

```
   string protectedFiscalCode = protector.Protect(fiscalCode);
   //TODO: qui persistiamo il valore di protectedFiscalCode nel db
   SaveFiscalCode(userId, protectedFiscalCode);
   return RedirectToAction(nameof(Index));
}
```

Il servizio `IDataProtectionProvider` dispone del metodo `CreateProtector`, che ci permette di creare un oggetto `IDataProtector` per uno specifico scopo, indicato come una stringa che va a integrare la chiave di cifratura principale. Per esempio, i dati cifrati per lo scopo "userData" non potranno essere decifrati da un `IDataProtector` creato per un altro scopo. In questo modo, rafforziamo l'isolamento tra i vari contenitori di dati gestiti dall'applicazione, che si rivela particolarmente utile anche per isolare in maniera granulare i dati tra tenant e tenant. Le applicazioni ASP.NET Core sono già automaticamente isolate le une dalle altre a prescindere dallo scopo utilizzato, perché ciascuna impiega un proprio spazio di archiviazione per le chiavi di cifratura.

Con il metodo `Protect` dell'oggetto `IDataProtector` possiamo cifrare sia stringhe sia array di byte. Il risultato cifrato avrà un aspetto simile a quello illustrato nella Figura 18.15.

```
25      [HttpPost]
        0 references
26      public IActionResult UpdateFiscalCode(string userId, string fiscalCode, [FromService
27      {
28      "CfDJ8DoPvoOA_bhCvyy79NA3n7tRyxP_bLE9k264WkZw72vjeqbjYBBHS6rKZgnj47F3m9akqdt-dRK8K8W9azmFrO2fGM
29      aBCzcTkTWiElBwm5gUlVW83PbzCg_jVz0FquypaA"
30          var protectedFiscalCode = protector.Protect(fiscalCode);
31          //...
32          return RedirectToAction(nameof(Index));
33      }
```

Figura 18.15 – Con il metodo `Protect` dell'oggetto `IDataProtector` otteniamo un contenuto cifrato.

Successivamente, quando sarà il momento di visualizzare i dati all'utente, dovremo per prima cosa decifrare il contenuto con il metodo `Unprotect`, come si può vedere nell'Esempio 18.38. Usiamo il blocco `try…catch` per catturare un'eventuale `CryptographicException`, che potrebbe verificarsi se i dati sono stati alterati in qualche modo.

Esempio 18.38

```
public IActionResult Show(string userId
                        [FromServices] IDataProtectionProvider provider) {
   //Usiamo lo stesso purpose usato all'atto della cifratura
   IDataProtector protector = provider.CreateProtector("userData");
   //TODO: qui otteniamo la stringa cifrata dal db
   string protectedFiscalCode = GetFiscalCodeFromDb(userId);
   string fiscalCode = null;
   //Decifriamo
   try {
      fiscalCode = protector.Unprotect(protectedFiscalCode);
   } catch (CryptographicException exc) {
      //TODO: logga l'eccezione
   }
   //Passiamo il valore a una view Razor per la visualizzazione
   var data = new UserData { FiscalCode = fiscalCode };
   return View(data);
}
```

Come abbiamo visto negli ultimi due esempi, è stato sufficiente invocare i metodi `Protect` e `Unprotect` dell'oggetto `IDataProtector` per cifrare o decifrare i dati. In questo caso semplice, non abbiamo dovuto indicare alcuna chiave di crittografia né configurare alcun algoritmo, perché di tutto questo si occupa la Data Protection API di ASP.NET Core. Proviamo ora ad addentrarci in questi meccanismi di gestione delle chiavi crittografiche, per capire in quali scenari è invece necessario fornire una configurazione personalizzata.

Gestire le chiavi crittografiche di Data Protection API

All'avvio dell'applicazione, ASP.NET Core usa tecniche euristiche per determinare quale sia il migliore spazio di archiviazione sulla piattaforma corrente per la conservazione delle chiavi crittografiche:

- Su Windows, le chiavi sono persistite nella directory del profilo dell'utente, al percorso `%localappdata%\ASP.NET\DataProtection-Keys`. Tuttavia, se l'applicazione gira inprocess con IIS, allora le chiavi

sono persistite in un ramo del registro HKLM di Windows, accessibile solo dal worker process dell'application pool. In entrambi i casi, le chiavi sono esse stesse protette con crittografia at-rest, grazie alla DPAPI di Windows.

- Su Linux e Mac troveremo le chiavi nella home directory dell'utente, per esempio nel percorso ~/.aspnet/DataProtection-Keys. A differenza di Windows, non saranno protette per default da alcuna crittografia at-rest ma è possibile abilitarla usando un certificato.
- Nei casi particolari in cui non sia possibile usare spazi di archiviazione persistenti, le chiavi verranno conservate in memoria e rigenerate al successivo riavvio dell'applicazione.

Conservare le chiavi sulla macchina locale non è sempre una situazione ideale. Negli scenari webfarm, infatti, l'applicazione funziona su più macchine server contemporaneamente ed è necessario che tutte condividano lo stesso spazio di archiviazione e usino le stesse chiavi di cifratura:

- Pubblicando nell'App Service di Azure non è necessario fare nulla, perché le chiavi sono persistite nella directory %HOME%\ASP.NET\DataProtection-Keys e l'infrastruttura si occupa di sincronizzarne il contenuto con tutte le istanze su cui viene eseguita l'applicazione.
- Se siamo noi a gestire la webfarm, dovremo indicare esplicitamente uno spazio di archiviazione condiviso, come Azure Blob Storage, Azure Key Vault, Redis o una semplice directory di rete.

Dal metodo ConfigureServices della classe Startup si può usare l'extension method AddDataProtection per configurare vari aspetti legati alla Data Protection API, tra cui lo spazio di archiviazione desiderato. L'Esempio 18.39 ci mostra come usare una directory condivisa in rete. Oltre a questo, bisogna ricordarsi anche di riabilitare la crittografia at-rest, perché selezionare esplicitamente uno spazio di archiviazione comporta la perdita delle euristiche: su Windows useremo ProtectKeysWithDpapi mentre su altre piattaforme sfrutteremo un certificato con ProtectKeysWithCertificate.

Esempio 18.39

```
//Windows
services.AddDataProtection()
   .PersistKeysToFileSystem(new DirectoryInfo(@"\\SRV1\Share"))
   .ProtectKeysWithDpapi() //Solo per Windows;
//Linux e Mac
var cert = new X509Certificate2("/var/cert.pfx", "password"); services.AddDataProtection()
   .PersistKeysToFileSystem(new DirectoryInfo("/mnt/share"))
   .ProtectKeysWithCertificate(cert); //Disponibile su tutte le piattaforme
```

Per una panoramica sugli spazi di archiviazione, possiamo leggere la documentazione Microsoft pubblicata all'indirizzo: http://aspit.co/bod.

La Data Protection API è ovviamente estendibile per supportare spazi di archiviazione personalizzati, implementando l'interfaccia IXmlRepository.

Un'altra questione di cui tener conto è la rigenerazione automatica delle chiavi crittografiche che, per default, avviene ogni 90 giorni. Questo accorgimento rappresenta una ulteriore linea difensiva in caso di compromissione delle chiavi. È possibile accelerare la rigenerazione impostando un numero di giorni arbitrario, fino a un minimo di 7 giorni, come è illustrato nell'Esempio 18.40.

Esempio 18.40

```
services.AddDataProtection()
          .SetDefaultKeyLifetime(TimeSpan.FromDays(7));
```

Le vecchie chiavi vengono comunque conservate dalla Data Protection API, in modo che sia possibile decifrare i contenuti protetti con esse. Per elencare, creare o revocare le chiavi presenti nello spazio di archiviazione, si può usare il servizio IKeyManager. Per ciascuna chiave dell'elenco, vengono esposte le date di inizio e fine validità e lo stato di revoca, come nella Figura 18.16.

```
public IActionResult Index([FromServices] IKeyManager keyManager) {
    IReadOnlyCollection<IKey> keys = keyManager.GetAllKeys();
    foreach (var key in keys) {
        key.
    }
```

ActivationDate DateTimeOffset ActivationDate
CreateEncryptor
CreationDate
Descriptor
Equals
ExpirationDate
GetHashCode
GetType
IsRevoked
KeyId
ToString

Figura 18.16 – Il servizio IKeyManager ci permette di interagire con le chiavi nello spazio di archiviazione.

Di solito non è necessario usare il servizio `IKeyManager` ma può risultare indispensabile nel momento in cui sia stata accertata una compromissione delle chiavi e sia perciò necessario cifrare i dati con nuove chiavi.

Infine, se non desideriamo usufruire della generazione automatica delle chiavi, possiamo rinunciarci usando l'extension method `DisableAutomaticKeyGeneration`, come è mostrato nell'Esempio 18.41.

Esempio 18.41

```
services.AddDataProtection()
.DisableAutomaticKeyGeneration();
```

Sarà quindi nostro compito generare almeno una chiave con il servizio `IKeyManager` oppure per mezzo di un'altra istanza dell'applicazione configurata per generare tale chiave.

Conclusioni

In questo capitolo abbiamo visto come affrontare l'importante tema della sicurezza nelle applicazioni web e quali sono gli strumenti messi a disposizione da ASP.NET Core per proteggere adeguatamente i dati dei nostri utenti. Con ASP.NET Core Identity possiamo creare un database locale completamente gestito dallo `UserManager<TUser>` e dalle altre API che si occupano delle operazioni di login e di protezione dai tentativi di intrusione. Usando la Data Protection API, possiamo proteggere stringhe in maniera on-demand per attuare la crittografia at-rest dei dati sensibili e mitigare così i problemi derivanti dal furto di dati, come disposto dalla normativa del GDPR.

Inoltre, durante le interazioni con l'applicazione web, gli utenti hanno la garanzia che i loro dati viaggino su una connessione sicura, con crittografia in-transit operata da un certificato SSL che troviamo già installato sin dalle fasi di sviluppo. I middleware di HTTPS redirection e HSTS impediscono che l'utente invii inavvertitamente una richiesta su un canale non sicuro durante la sua navigazione. L'utilizzo corretto dell'applicazione è anche assicurato da ASP.NET Core MVC, che impiega diverse tecniche di prevenzione dei più comuni tipi di attacco HTTP.

Microsoft include questi accorgimenti in ogni nuova applicazione ASP.NET Core creata da template, in modo che sia sicura di default.

Dal prossimo capitolo volgeremo invece lo sguardo verso la parte client dell'applicazione, per capire come costruire una migliore user experience usando JavaScript.

19
Utilizzare JavaScript in applicazioni ASP.NET

Finora abbiamo parlato di argomenti che riguardano lo sviluppo lato server cioè di come generare dati da inviare al client a seguito di una sua richiesta e di come processare dati che provengono dal client. Ora cambiamo approccio e cominciamo a occuparci delle tecnologie lato client e di come queste interagiscano con il server.

Da quando il JavaScript è stato fortemente arricchito di funzionalità, le applicazioni web si sono divise in due grandi categorie: le applicazioni ibride e le Single Page Application (SPA). Le prime sono applicazioni dove il flusso prevede che il client richieda una pagina e il server risponda con la pagina. Il client esegue l'eventuale codice JavaScript nella pagina e quando ha bisogno di una nuova pagina la richiede al server ricominciando il flusso. Le seconde sono applicazioni interamente sviluppate in HTML e JavaScript e CSS dove l'applicazione gira interamente sul client, l'unica interazione con il server è per recuperare i dati (generalmente in formato JSON).

ASP.NET Core MVC offre supporto completo per entrambe i tipi di applicazione, ma per sua natura si sposa alla perfezione con le applicazioni ibride, in quanto il client richiama un url e ASP.NET Core MVC risponde con codice HTML che il client renderizza e di cui esegue l'eventuale codice JavaScript. ASP.NET Core Web API, al contrario, si sposa perfettamente con le SPA, in quanto è un framework pensato esclusivamente per creare API che forniscono dati al client.

L'argomento di questo capitolo sarà l'utilizzo di JavaScript con applicazioni ibride mentre nel Capitolo 21 affronteremo le SPA. Più in dettaglio, nelle prossime sezioni affronteremo le librerie JavaScript che sono maggiormente utilizzate per questo tipo di applicazioni: jQuery, Bootstrap e Vue.js (o semplicemente Vue). Un'altra libreria molto utilizzata è AngularJS. Tuttavia Google ha dichiarato che non evolverà più AngularJS al di là di patch di sicurezza, quindi non parleremo di questa libreria in quanto possiamo anticipare che la sua adozione calerà drasticamente in futuro. Il motivo per cui Google non evolverà più AngularJS è che manterrà solo Angular, una nuova versione pensata esclusivamente per realizzare SPA, di cui parleremo nel Capitolo 21.

Prima di cominciare a parlare in dettaglio delle singole librerie, vediamo come aggiungerle alla nostra applicazione e come gestirne gli aggiornamenti attraverso i tool di Visual Studio.

Gestire le librerie JavaScript

Nel corso del libro abbiamo utilizzato NuGet per referenziare librerie come quelle di .NET Core, di ASP.NET Core e di altri framework di terze parti. Sebbene in NuGet siano presenti anche librerie client, a partire da Visual Studio 2017, Microsoft consiglia l'utilizzo di NuGet solamente per referenziare librerie .NET lasciando ad altri strumenti il compito di referenziare quelle client.

Inizialmente, questo compito veniva svolto da un tool che si interfacciava con Bower che è un repository pubblico appositamente pensato per contenere librerie JavaScript. Tuttavia, con l'esplosione di NodeJS e del suo repository NPM, il team di Bower ha dichiarato che non evolverà più il repository, ma lo manterrà solo per retro compatibilità. Per questo motivo, al momento della stesura di questo libro, con la versione 15.8 di Visual Studio il team ha intenzione di introdurre un nuovo strumento per gestire le librerie JavaScript: LibMan.

LibMan è basato su un file di nome `libman.json` che va aggiunto nella root del progetto cliccando col tasto destro e selezionando la voce Manage Client-Side Libraries dal menu contestuale visibile nella Figura 19.1.

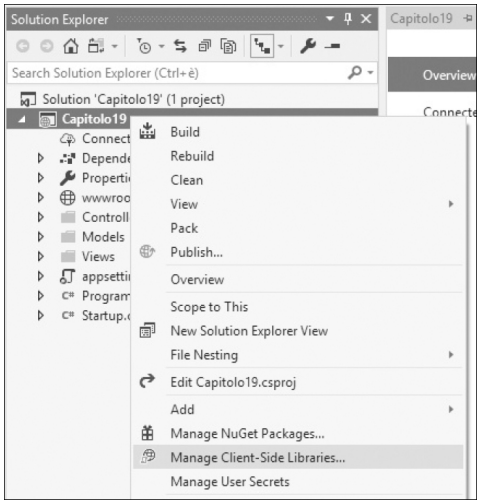


Figura 19.1 – La voce di menu per aggiungere LibMan al progetto.

LibMan è pensato per non essere specifico su un repository, quindi è stato disegnato con un'architettura a provider che gli consente di andare su più repository a seconda della scelta dell'utente. Attualmente, LibMan supporta due provider: cdnjs, che si connette all'omonimo repository, e filesystem che recupera i file da un percorso del file system. Il provider è specificato nella proprietà del `defaultProvider` del file. Visual Studio offre già l'autocomplete per impostare questa proprietà, come dimostra la Figura 19.2.

Figura 19.2 – Il meccanismo di autocomplete per il file `libman.json`.

Un'altra proprietà del file è `version`. Questa specifica la versione del file ed è importante per Visual Studio per capire a quale versione di LibMan fare riferimento. Al momento il solo valore ammesso per questa proprietà è `1.0`.

La proprietà `defaultDestination` specifica il percorso della cartella all'interno della quale inserire i file scaricati. Il percorso della cartella deve essere espresso relativamente alla root del progetto. L'ultima proprietà da impostare è `libraries` ed è la più importante in quanto specifica quali librerie vogliamo scaricare; questa proprietà è l'unica obbligatoria del file `libman.json`. `Libraries` contiene un array di oggetti dove ogni oggetto rappresenta una libreria. Ogni oggetto ha le proprietà `library`, `provider`, `destination` e `files`.

La proprietà `library` specifica il nome e la versione della libreria da scaricare nel formato `nome@versione`. La proprietà `provider` specifica il provider da utilizzare. Se non abbiamo specificato la proprietà `defaulProvider`, `provider` è obbligatoria, mentre in caso contrario, `provider` sovrascrive il valore di `defaulProvider`. La proprietà `destination` specifica la cartella in cui copiare i file. Anche questa proprietà è obbligatoria solo se non abbiamo specificato il default tramite `defaultDestination` e sovrascrive il default se presente. La proprietà `files` specifica quali file della libreria vogliamo scaricare. Infatti, molte librerie mettono a disposizione i file originali, i file minificati, i file di documentazione e così via. Tramite la proprietà `files` possiamo impostare un'array con i nomi dei file che vogliamo scaricare. Il prossimo codice mostra un esempio di file libman.json che referenzia jQuery, Bootstrap e Vue.js.

Esempio 19.1

```
{
   "version": "1.0",
   "defaultProvider": "cdnjs",
   "libraries":
   [
      {
         "library": "jquery@3.3.1",
         "destination": "wwwroot/lib/jquery",
         "files": ["jquery.slim.min.js" ]
      },
      {
         "library": "twitter-bootstrap@4.1.1",
         "destination": "wwwroot/lib/bootstrap/"
      },
      {
         "library": "vue@2.5.16",
         "destination": "wwwroot/lib/vue/"
      }
   ]
}
```

Nella Figura 19.2 abbiamo visto che Visual Studio offre l'intellisense per la proprietà `defaultProvider`, ma questa non è l'unica. Infatti, abbiamo a disposizione l'intellisense per tutte le proprietà del file e questo torna molto utile soprattutto quando dobbiamo compilare la proprietà `library`. Infatti, dopo che abbiamo cominciato a digitare le prime lettere del nome del pacchetto, Visual Studio si connette al repository e scarica la lista di pacchetti che iniziano con le lettere che abbiamo inserito. Una volta selezionata la libreria e aggiunto il carattere @ Visual Studio abilita l'intellisense anche per le versioni della libreria offrendo così un supporto completo. Una volta selezionate la libreria e la versione, abbiamo a disposizione l'intellisense anche per la proprietà `files`, in quanto Visual Studio si connette di nuovo al repository e scarica i nomi dei file della libreria per poi farli selezionare. Ora che abbiamo visto come referenziare le librerie, vediamo come usarle nel nostro codice a partire da jQuery.

jQuery

jQuery è una libreria JavaScript molto potente, che mette a disposizione degli sviluppatori funzionalità avanzate per la ricerca, la navigazione e la manipolazione del DOM, per la gestione degli eventi, per la gestione delle richieste AJAX, delle promise e altro ancora.

Maggiori informazioni su jQuery sono disponibili all'url: http://aspit.co/a10.

Uno dei capisaldi di jQuery è stato quello di aggiungere funzionalità che ci permettono di scrivere la minor quantità di codice possibile. Questo risultato è stato raggiunto sfruttando alcune tecniche come la sintassi fluent, la ricerca nel DOM attraverso selettori CSS3 e l'utilizzo di nomi di metodi molto brevi ma intuitivi. Cominciamo a scoprire le singole funzionalità di jQuery partendo dalla capacità di effettuare ricerche nel DOM.

Effettuare ricerche nel DOM

jQuery è nato per eseguire query nel DOM del browser al fine di recuperare gli elementi HTML al suo interno. Queste query sono effettuate basandosi sulla sintassi dei selettori CSS, esattamente come avviene per il metodo `querySelectorAll` dell'oggetto `document`.

`querySelectorAll` è un metodo introdotto da HTML5 e quindi relativamente giovane; jQuery permetteva di effettuare query già nel 2007.

Per sfruttare jQuery, il punto di entrata è l'oggetto `$`. Questo oggetto può essere usato sia come metodo sia come oggetto. Se vogliamo effettuare una query nel DOM, sfruttiamo questo oggetto come metodo passando in input la stringa di ricerca, così come nel prossimo esempio.

Esempio 19.2

```
//ricerca un elemento per id
const objects = $('#textboxId');

//ricerca un elemento per id fornito da ASP.NET
const objects = $('#@Html.IdFor(m => m.Name)');

//ricerca elementi per classe
const objects = $('.classe');

//ricerca tag p all'interno dei div con classe "contenitore"
const objects = $('div.contenitore p');
```

Ci sono tre cose da notare in questo esempio. La prima è che, a prescindere da quanti oggetti possa tornare la query, il metodo da usare è sempre lo stesso. La seconda è che possiamo facilmente interpolare codice JavaScript e codice ASP.NET come il secondo metodo dimostra. La terza è che la variabile `objects` non contiene una lista di oggetti HTML, come ci si potrebbe aspettare, bensì un oggetto di jQuery (detto contenitore) che contiene la lista degli oggetti recuperati al suo interno. Questo oggetto dimostra la sua utilità proprio quando dobbiamo manipolare gli oggetti al suo interno.

Manipolare gli oggetti

Supponiamo di voler aggiungere una classe CSS a tutti gli elementi restituiti da una query. Se non utilizzassimo jQuery, una volta ottenuta la lista di oggetti dovremmo eseguire un ciclo, aggiungendo la classe a tutti. Con jQuery possiamo fare tutto in una riga di codice, invocando il metodo `addClass` del contenitore passandogli in input il nome della classe, come viene mostrato nell'Esempio 19.3.

Esempio 19.3

```
const objects = $('.myClass')
   .addClass('selected')
   .addClass('anotherClass');
```

Il metodo `addClass` scorre tutti gli oggetti nel contenitore e aggiunge a ognuno di essi la classe `selected`, permettendoci quindi di risparmiare codice. Non solo, il metodo `addClass` è una funzione che ritorna il contenitore stesso permettendoci di sfruttare la tecnica fluent per invocare di nuovo il metodo `addClass` per aggiungere un'altra classe CSS agli oggetti. Il metodo `addClass` è solo uno dei tanti metodi per manipolare gli oggetti: parlare di tutti i metodi è impossibile per ovvie questioni di spazio, quindi vi offriamo un riepilogo di quelli più importanti nella Tabella 19.1.

Tabella 19.1 – Metodi del contenitore jQuery per manipolare il DOM.

Metodo	Descrizione
`append`	Prende in input un oggetto jQuery o una stringa HTML e li aggiunge al contenuto di tutti gli oggetti nel contenitore.
`html`	Se non passiamo alcun parametro, ritorna la proprietà innerHTML del primo oggetto nel contenitore. Se passiamo una stringa, la imposta come proprietà innerHTML di ogni oggetto nel contenitore.
`text`	Si comporta esattamente come il metodo html, con la differenza di utilizzare la proprietà innerText.
`after before`	Accettano in input un oggetto jQuery o una stringa HTML, che vengono aggiunti rispettivamente prima o dopo ogni oggetto nel

	contenitore.
empty	Elimina il contenuto di ogni oggetto nel contenitore.
remove	Accetta in input una lista di oggetti da rimuovere dal DOM.
attr	Accetta in input il nome di un attributo HTML e restituisce il valore di quell'attributo per il primo oggetto nel contenitore. Se il contenitore contiene più oggetti, gli altri vengono ignorati. Questo metodo ha un overload che accetta un secondo valore. In questo caso, l'attributo HTML viene impostato con il secondo valore.
removeAttr	Accetta in input il nome dell'attributo HTML da eliminare dal primo oggetto nel contenitore.
val	Ritorna la proprietà value del primo oggetto nel contenitore. Questo metodo ha un overload che accetta un valore che viene poi usato per impostare la proprietà value del primo oggetto nel contenitore.
addClass removeClass, toggleClass	I primi due metodi aggiungono e rimuovono rispettivamente una classe CSS dagli oggetti nel contenitore. Il terzo metodo esegue una verifica sull'esistenza della classe CSS sull'oggetto: se è già presente, allora la rimuove, altrimenti la aggiunge.
hide, show, toggle	I primi due metodi mostrano e nascondono rispettivamente gli oggetti nel contenitore. Il terzo metodo esegue una verifica sulla visibilità dell'oggetto: se è visibile, allora lo nasconde, altrimenti lo rende visibile.

Oltre a manipolare gli oggetti, jQuery permette anche di gestirne gli eventi.

Gestire gli eventi

Gestire gli eventi degli oggetti con jQuery è semplice come manipolare gli oggetti stessi. Una volta recuperati gli oggetti che vogliamo monitorare e manipolare, dobbiamo usare il metodo on passando in input il nome dell'evento e il delegato da invocare quando si scatena l'evento.

Se invece vogliamo eliminare la sottoscrizione all'evento, dobbiamo usare il metodo off, che accetta gli stessi parametri di on. L'utilizzo di on e off è visibile nel prossimo esempio.

Esempio 19.4

```
//si sottoscrive all'evento click di un pulsante
$('#myButton').on('click', handler);

//si cancella dall'evento click di un pulsante
$('#myButton').off('click', handler);

function handler() {
   alert('bottone cliccato');
}
```

Oltre a sottoscriverci, possiamo anche scatenare programmaticamente un evento tramite il metodo trigger, che accetta in input il nome dell'evento da scatenare.

C'è un evento particolare in jQuery che merita attenzione, cioè l'evento scatenato quando la pagina è caricata dal browser. Per intercettare questo evento, ci basta passare a $ una funzione invece che una stringa. In questo caso, il codice della funzione viene eseguito nel momento del caricamento della pagina.

Quella di lavorare con gli oggetti del DOM è stata la prima funzionalità di jQuery ma col tempo se ne sono aggiunte altre, la più importante delle quali è senz'altro la possibilità di effettuare chiamate AJAX.

Effettuare chiamate AJAX

Eseguire una chiamata AJAX utilizzando direttamente XMLHttpRequest è semplice ma richiede una notevole quantità di codice rispetto a quello che dobbiamo scrivere usando jQuery. jQuery mette a disposizione il metodo ajax, il quale accetta in input un oggetto che contiene le proprietà necessarie per effettuare la chiamata. Le proprietà fondamentali di questo oggetto sono elencate nella Tabella 19.2.

Tabella 19.2 – Principali proprietà dell'oggetto in input al metodo ajax di jQuery.

Proprietà	Descrizione
cache	Se impostato a false disabilita la cache del browser, appendendo un timestamp all'url.
data	Rappresenta un oggetto contenente i dati da inviare al server. Se la richiesta è di tipo GET, l'oggetto viene serializzato in querystring, altrimenti viene serializzato nel corpo della richiesta.
headers	Rappresenta un oggetto contenente le intestazioni da aggiungere alla richiesta.

jsonp	Se impostata, specifica che la chiamata deve sfruttare la tecnica JSONP e il valore della proprietà rappresenta il metodo da invocare a fine chiamata.
timeout	Specifica il timeout della richiesta in millisecondi.
type	Specifica il tipo della richiesta (POST, GET e così via).
url	Specifica l'url da invocare.
contentType	Specifica il tipo di dati inviati nel corpo della richiesta. Di default è impostato su application/x-www-form-urlencoded; charset=UTF-8 ma, per oggetti JSON, dobbiamo impostarlo su applicaton/json.

Poiché le chiamate AJAX sono asincrone, il metodo `ajax` ritorna una promise di jQuery che è diversa da una promise JavaScript. Una promise jQuery è un'oggetto di tipo `Deferred` che rappresenta un'operazione in sospeso (in questo caso la chiamata) e che, una volta terminata, invoca determinati metodi per sfruttarne il risultato. I metodi sono:

- `done`: invocato quando la promise torna con successo;
- `fail`: invocato quando la promise torna un errore (per esempio, quando il server torna un errore);
- `always`: invocato a prescindere dal risultato della promise.

L'Esempio 19.5 mostra come usare il metodo `ajax`.

Esempio 19.5

```
$.ajax({
   url: '/userhandler'
   data: { userId: 1 }
})
.done(function(result){
   alert(result);
})
.fail(function(){
   alert('errore');
})
.always(function(){
   alert('chiamata terminata');
});
```

Il metodo `ajax` è banale ma jQuery offre anche dei metodi, basati su `ajax`, che semplificano ancora di più il codice. Questi metodi sono: `get`, `getJSON`, `post` e `load`.

Il metodo `get` esegue una chiamata GET e restituisce il risultato come stringa. Il metodo `getJSON` esegue una chiamata GET e si aspetta come risposta dal server una stringa JSON che successivamente viene trasformata in un oggetto JavaScript che viene poi restituito al chiamante. Il metodo `post` esegue una chiamata POST e restituisce il risultato al chiamante. Infine il metodo `load` esegue una chiamata GET caricando il risultato nell'oggetto del DOM su cui si applica il metodo.

Tutti i metodi accettano in input l'url da invocare e i parametri da passare, e ritornano una promise. Il loro utilizzo è visibile nell'Esempio 19.6.

Esempio 19.6

```
$.get('/userhandlerget', { userId: 1 });

$.getJSON('/userhandlerjson', { userId: 1 });

$.post('/userhandlerpost', { userId: 1 });

$('#divid').load('/userhandlerload', { userId: 1 });
```

Dagli esempi che abbiamo visto in questa sezione, è evidente come jQuery semplifichi notevolmente il codice JavaScript da scrivere per eseguire le operazioni basilari compiute da ogni applicazione che abbia un minimo di interazione client. Oltre a questo, jQuery mette a disposizione anche una libreria di controlli visuali già pronti per essere utilizzati nella nostra applicazione: jQueryUI.

jQueryUI

Per sfruttare al massimo la semplicità con cui jQuery permette di lavorare con il DOM, il team di jQuery ha creato una libreria di componenti visuali chiamata jQueryUI. Allo stato attuale, la libreria comprende 14 controlli visuali, ma quelli più usati sono i seguenti:

- `accordion`: racchiude un insieme di pannelli mostrandone solo le intestazioni; il contenuto di un solo pannello per volta per volta può essere visibile;
- `autocomplete`: permette l'autocomplete sulle textbox;
- `datepicker`: mostra un calendario associato a una textbox;
- `dialog`: mostra una finestra di dialogo in overlay;
- `tabs`: racchiude pannelli in una visualizzazione a tab.

Non parleremo in dettaglio di questi componenti ma, nel prossimo esempio, mostreremo quanto sia semplice il loro uso.

Esempio 19.7

```
//html
<input type="text" id="datepicker">

//JavaScript
$(function() {
   $('#datepicker').datepicker();
});
```

In questo esempio dichiariamo una textbox nel codice HTML e, successivamente al caricamento della pagina, le associamo il controllo `datepicker`. Il risultato è che quando la textbox prende il fuoco, viene visualizzato il calendario come nella Figura 19.3.

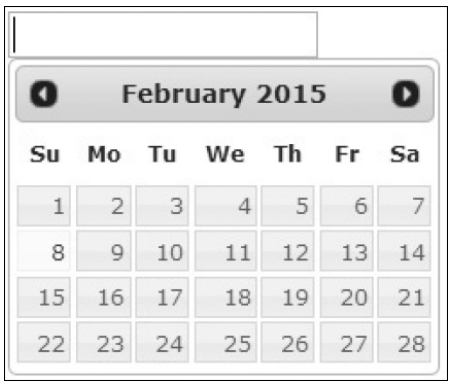


Figura 19.3 – Il controllo datepicker in azione.

Tutti i controlli di jQueryUI lavorano con codice HTML strutturato in maniera precisa, quindi è bene leggere la relativa documentazione sul sito, disponibile all'indirizzo: `http://aspit.co/ain`.

> *jQuery mette a disposizione anche delle animazioni che scattano quando si mostra o nasconde un oggetto o quando si aggiunge o rimuove una classe CSS. Inoltre, mette a disposizione anche dei comportamenti per arricchire la nostra applicazione, come il drag&drop, l'ordinamento di oggetti e così via.*

jQuery è oramai una realtà consolidata del web in quanto è presente da ormai oltre 10 anni e il suo utilizzo è molto diffuso. Moltissimi framework e librerie per lo sviluppo web si basano su jQuery e uno di questi è Bootstrap.

Bootstrap

Bootstrap è una libreria web che ci offre una serie di classi CSS e di controlli visuali JavaScript già pronti per l'uso. Questa libreria è stata creata ed è manutenuta dal team di Twitter, il che la rende affidabile agli occhi degli sviluppatori.

La potenza di Bootstrap risiede nel fatto che le classi CSS, in combinazione con il codice HTML impostato secondo il loro scopo, coprono gran parte delle necessità di un'applicazione web, come il layout responsive, gli stili di intestazioni, paragrafi, testo, bottoni, icone e molto altro ancora. Grazie all'estrema semplicità con cui si possono creare layout, Bootstrap è al momento una delle migliori librerie HTML/CSS in circolazione.

> *Al momento della scrittura di questo libro, Bootstrap è giunto alla versione 4.1.1 ed è disponile all'indirizzo http://aspit.co/a1y.*

I controlli visuali utilizzano JavaScript per creare tab, tooltip, finestre modali e altro ancora. Questi controlli sono basati su jQuery e sono un'alternativa a jQueryUI. jQueryUI e Bootstrap hanno alcuni controlli in comune e altri diversi. La scelta di usare i componenti di una libreria o dell'altra è dettata dalle esigenze del progetto e dalle conoscenze personali.

I controlli più utilizzati di Bootstrap sono:

- `Alert`: mostra un pannello con un messaggio; utile per fornire un feedback su un'azione dell'utente;
- `Carousel`: crea uno slideshow;
- `Collapse`: mostra o nasconde porzioni di HTML; permette anche di creare un accordion;
- `Modal`: mostra una finestra modale;
- `Navbar`: crea un menu orizzontale con voci e sottovoci su più livelli;
- `Tooltip`: sostituisce il tooltip standard del browser con uno custom personalizzabile da CSS;
- `Tab`: crea una visualizzazione dei contenuti a tab.

La maggior parte di questi componenti può essere utilizzata senza dover scrivere codice JavaScript ma semplicemente generando il codice HTML con tag e attributi specifici elencati nella documentazione.

Grazie a questi tag e attributi, Bootstrap si mette automaticamente in ascolto di eventi e li gestisce per noi.

Il JavaScript di Bootstrap è incluso in un solo file, ma ha una dipendenza da una libreria di terze parti chiamata popper. Questa non è distribuita da Bootstrap e va quindi inclusa negli script della pagina.

Prendiamo come esempio il controllo Alert. Questo controllo mostra un messaggio che informa l'utente di un evento e permette di visualizzare un pulsante di chiusura del messaggio. Il codice necessario a mostrare il messaggio è visibile nel prossimo esempio.

Esempio 19.8

```
<div class="alert alert-success alert-dismissible" id="message">
   <strong>Terminato!</strong> Hai completato la registrazione con successo.
   <button type="button" class="close" data-dismiss="alert">
      <span>&times;</span>
   </button>
</div>
```

Il messaggio viene visualizzato in un box stilizzato usando le classi CSS `alert`, `alert-success` e `alert-dismissible`. Il pulsante viene mostrato sulla destra del box. Il risultato è visibile nella Figura 19.4.

Figura 19.4 – il controllo Alert di bootstrap con testo e tasto di chiusura.

Quando clicchiamo sul pulsante di chiusura, grazie all'attributo `data-dismiss` e al suo valore `alert` l'intero box sparisce. Questo accade perché Bootstrap si mette in ascolto sul click degli oggetti con l'attributo `data-dismiss` e se il valore è `alert`, ne nasconde il box che li contiene.

Oltre all'utilizzo dell'attributo, possiamo anche utilizzare il codice JavaScript per chiudere il box tramite il metodo `close` del plugin `alert` che rappresenta il controllo. Non solo, se dobbiamo eseguire logica prima o dopo la chiusura del box abbiamo anche a disposizione gli eventi `close.bs.alert` e `closed.bs.alert` su cu metterci in ascolto. L'Esempio 19.9 mostra come sfruttare il controllo Alert via JavaScript.

Esempio 19.9

```
$('#message').alert('close'); //chiude l'alert

$('#message).on('close.bs.alert', function () {
   console.log('In chiusura');
});

$('#message).on('closed.bs.alert', function () {
   console.log('chiuso');
});
```

Un altro controllo molto usato in Bootstrap è Modal per mostrare una finestra modale. Questo controllo segue esattamente le stesse regole viste con Alert. Se abbiamo un comportamento standard, (apertura e chiusura con pulsanti con specifici attributi) Modal può essere

gestito da codice HTML senza alcuna necessità di codice JavaScript. Se invece abbiamo la necessità di personalizzare il comportamento, per esempio, dobbiamo aprire la modale solo a seguito di alcuni controlli, abbiamo a disposizione il plugin JavaScript `modal` con metodi ed eventi per gestire il controllo.

Queste stesse considerazioni si applicano a tutti i controlli che Bootstrap mette a disposizione. La documentazione sul sito è molto vasta e ben strutturata quindi rimandiamo a questa per i dettagli su ogni singolo controllo.

JQuery si concentra sulla semplificazione nel manipolare il DOM, jQueryUI si concentra sul creare controlli riutilizzabili e Bootstrap si concentra sul semplificare il codice HTML e la sua stilizzazione e sulla creazione di componenti. Tutte queste librerie si basano sull'utilizzo spinto del DOM. Vediamo ora una libreria che ci fa lavorare molto meno con il DOM.

Vue.js

Con l'introduzione del paradigma AJAX, c'è stato sempre un maggior sviluppo di funzionalità sul client e quindi una sempre maggior necessità di codice JavaScript da scrivere. Per ridurre la quantità di codice, si possono utilizzare librerie che introducono il concetto di binding.

Queste librerie permettono di legare gli oggetti nella pagina con le proprietà di un oggetto JavaScript (definito viewmodel, controller, sorgente o in altri modi diversi, a seconda della libreria; noi useremo viewmodel) e di mantenere i valori aggiornati tra loro. Supponiamo di avere un campo di testo legato a una proprietà del viewmodel. Tramite queste librerie possiamo modificare la proprietà e far sì che il campo di testo nella pagina sia aggiornato e, viceversa, fare in modo che, qualora l'utente modifichi il campo di testo, il valore della proprietà sia aggiornato: questo sistema di collegamento tra viewmodel e oggetti nella pagina viene definito binding e il pattern di sviluppo che lo sfrutta viene definito Model-View-ViewModel (MVVM).

Esistono molte librerie che permettono questa funzionalità e tra le più utilizzate ci sono sicuramente Vue, AngularJS e Knockout. Knockout è stata una delle prime librerie a offrire il binding sul client ed è tuttora un progetto manutenuto ed evoluto. Tuttavia, la crescita di AngularJS negli anni ne ha ridotto l'utilizzo da parte degli sviluppatori. AngularJS ha vissuto un autentico boom intorno alla metà di questo decennio grazie alla sua semplicità e al fatto che dietro il suo sviluppo ci fosse Google. Con l'uscita del suo successore (Angular, di cui parleremo nel Capitolo 21) questa libreria ha vissuto un declino e Google ha dichiarato che creerà un'ultima release e poi rilascerà solo aggiornamenti per risolvere bug, al fine di non mettere in pericolo i numerosi progetti basati su questa tecnologia.

Questo ha portato molti sviluppatori a considerare Vue per le loro applicazioni. Vue è una libreria che offre il binding, ma che, grazie a una serie di librerie aggiuntive, permettere di creare anche Single Page Application. La sua semplicità di apprendimento, le sue somiglianze con AngularJS e le sue performance hanno permesso a molti sviluppatori di migrare a questa libreria molto velocemente. In questa sezione mostreremo un esempio di binding usando Vue.

Il sito di Vue è disponibile a questo indirizzo: http://aspit.co/bnd.

Il primo passo per usare Vue è includere il file JavaScript nella pagina. Una volta fatto questo, per stabilire il binding dobbiamo creare il viewmodel che espone le proprietà che vogliamo collegare alla UI. Il viewmodel è un oggetto di tipo `Vue` che espone una proprietà `el`, che contiene l'elemento HTML su cui vogliamo applicare il binding (come selettore CSS o come oggetto del DOM) e una proprietà `data` che contiene un oggetto con le proprietà da mettere in binding. Infine, dobbiamo creare il codice HTML, aggiungendo le informazioni che lo mettono in binding con l'oggetto JavaScript. Nell'Esempio 19.10, vediamo il codice necessario a eseguire queste operazioni.

Esempio 19.10

```
<div id="binding" style="border: 1px solid black">
   <span>{{ message }}</span>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
   const app = new Vue({
      el: '#binding',
      data: {
         message: 'messaggio via binding'
      }
   });
</script>
```

Nel codice HTML, la sintassi che prevede l'uso delle parentesi graffe doppie specifica che all'interno di esse c'è un'espressione JavaScript che Vue valuta a runtime e il cui risultato verrà usato per sostituire l'espressione stessa e le parentesi graffe. Il risultato dell'Esempio 19.10 è quello mostrato nella Figura 19.5.

messaggio via binding

Figura 19.5 – La pagina dopo il binding con Vue.

Se modifichiamo il valore della proprietà `message` nel viewmodel, il nuovo valore verrà mostrato nella pagina, mentre invece non c'è modo di aggiornare il valore della proprietà nel viewmodel dalla pagina; in questi casi, si parla di binding one-way. Quando lavoriamo con i campi di input di una form, invece, abbiamo anche la possibilità di aggiornare dalla pagina la proprietà nel viewmodel; in questi casi, si parla di binding two-way.

Vue permette di sfruttare il binding two-way (tecnicamente non è un binding two-way ma il risultato è lo stesso) tramite la direttiva `v-model` che prende in input il nome della proprietà che deve essere aggiornata dal campo di input. Una direttiva è un attributo HTML che Vue riconosce e che utilizza per i propri scopi.

Esempio 19.11

```
<input type="text" v-model="name" />
{{name}}
<script>
   const app = new Vue({
      el: '#binding',
      data: {
         name: 'Stefano Mostarda'
      }
   });
</script>
```

In questo esempio, ogni volta che modifichiamo il testo del campo input, questo cambiamento verrà riportato nella proprietà `name` del viewmodel, che a sua volta aggiornerà tramite binding one-way l'espressione tra parentesi graffe.

Il binding one-way e quello two-way possono essere combinati per creare anche effetti più complessi. Nell'Esempio 19.12 sfruttiamo queste funzionalità insieme alla direttiva `v-bind` per abilitare un pulsante e nascondere un div a seconda della selezione di una checkbox.

Esempio 19.12

```
<div id="binding">
   <input type="checkbox" v-model="selected" />check
   <button v-bind:disabled="!selected">button</button>
   <div v-bind:class="{ 'd-none': !selected }">Contenuto</div>
</div>

<script>
   const app = new Vue({
      el: '#binding',
      data: {
         selected: true
      }
   });
</script>
```

La sintassi della direttiva `v-bind` prevede che dopo i due punti si metta in binding un attributo HTML, al quale vogliamo assegnare il valore della direttiva. In questo caso, l'attributo `disabled` viene impostato solo se la proprietà `selected` è uguale a `false`. Nel caso della classe CSS, la sintassi si arricchisce ulteriormente. Potendo assegnare più classi all'attributo `class`, in input alla direttiva non passiamo una stringa ma un oggetto, dove ogni proprietà rappresenta il nome della classe CSS che vogliamo applicare, mentre il suo valore specifica se dobbiamo applicare quella classe CSS o no (true o false). Il risultato sarà che la classe CSS `d-none` (che nasconde il div) verrà applicata solo se la checkbox non è selezionata.

Nascondere il div è una delle opzioni, ma potremmo anche non renderizzarlo nel DOM utilizzando la direttiva `v-if` che, se impostata con una proprietà che ha valore `false`, fa esattamente questo. Oltre a mettere in binding proprietà che hanno un valore singolo, possiamo mettere in binding una lista di oggetti per disegnare una griglia o popolare gli elementi di un tag select o altro ancora. Per queste esigenze, Vue mette a disposizione la direttiva `v-for` che permette di replicare un tag HTML per ogni elemento della lista di input.

Esempio 19.13

```
<div id="binding">
   <ul>
      <li v-for="item in people">
         {{ item.firstName }} {{ item.lastName }}
      </li>
         </ul>
</div>

<script>
   const app = new Vue({
      el: '#binding',
      data: {
         people: [
            { firstName: 'Stefano', lastName: 'Mostarda' },
            { firstName: 'Daniele', lastName: 'Bochicchio' }
         ]
      }
   });
</script>
```

In questo esempio, il tag `li` viene ripetuto per ogni elemento nella proprietà `people` e viene usata la sintassi di binding one-way per mostrare le proprietà di ogni singolo oggetto.

Il pattern MVVM non è utile solo per mostrare i dati a video ma anche per gestire gli eventi scatenati dagli oggetti, come il click di un pulsante, il mouse over di un oggetto o altro ancora. Vue mette a disposizione la direttiva `v-on` per mettere in binding un evento sulla UI con un metodo del viewmodel, così da permetterci di manipolare i dati a seguito di un evento. Il metodo del viewmodel deve essere dichiarato all'interno della proprietà `methods` come viene mostrato nel prossimo esempio.

Esempio 19.14

```
<div id="binding">
   <button v-on:click="showMessage">show message</button>
   <span v-if="show">Messaggio visibile</span>
</div>

<script>
   const app = new Vue({
      el: '#binding',
      data: {
         show: false
      },
      methods: {
         showMessage: function () {
            this.show = true;
         }
      }
   });
</script>
```

La sintassi della direttiva `v-on` prevede che dopo il nome della direttiva venga inserito il separatore `:` (due punti) seguito dal nome dell'evento. Il valore della direttiva è il nome del metodo da invocare nel viewmodel. Questo metodo può accedere a tutte le variabili del viewmodel semplicemente utilizzando la parola chiave JavaScript `this`.

Vue.js e ASP.NET

Nella sezione precedente abbiamo utilizzato codice HTML che non ha nulla a che fare con ASP.NET ma che può essere generato partendo da un tag helper. Questo significa che l'integrazione tra Vue e ASP.NET dal punto di vista del codice HTML è estremamente semplice in quanto dobbiamo semplicemente aggiungere gli attributi di Vue al nostro tag helper.

Dal punto di vista del codice JavaScript, dobbiamo creare il codice del viewmodel e inserirci i dati di binding, perché non basta, per esempio, impostare l'attributo `value` del campo di input, ma dobbiamo impostare anche il valore del campo in binding nel viewmodel di Vue. Nel prossimo esempio vediamo come creare un viewmodel JavaScript sfruttando dati provenienti dal model della view.

Esempio 19.15

```
<div id="binding">
   <input type="text" asp-for="Name" v-model="name" />
   {{name}}
```

```
</div>

<script>
   const app = new Vue({
      el: '#binding',
      data: {
         name: '@Model.Name'
      }
   });
</script>
```

In questa sezione abbiamo scoperto come grazie a binding, scriviamo codice HTML e JavaScript che si occupa principalmente del business e non dei dettagli del DOM, creando funzionalità interessanti con poche righe di codice. Queste funzionalità nel corso degli anni sono sempre più migliorate e hanno spinto l'utilizzo di JavaScript e HTML a un livello ancora più avanzato, tanto da permettere di creare applicazioni che girano interamente sul client: le Single Page Application.

Conclusioni

In questo capitolo abbiamo visto come introdurre JavaScript nei nostri progetti grazie a LibMan. Questa libreria, infatti, rende semplicissimo il compito di recuperare librerie di terze parti, scaricare i file necessari e piazzarli esattamente nelle cartelle dove vogliamo siano salvati. Inoltre, in qualunque momento, possiamo aggiornare le librerie a una nuova versione, sempre modificando il file `libman.json`.

Successivamente, abbiamo visto come librerie come jQuery, jQueryUI, Bootstrap e Vue possano semplificare notevolmente la scrittura di codice JavaScript o, in alcuni casi (come con Bootstrap), eliminarla del tutto e utilizzare semplice codice HTML. In applicazioni ibride, queste librerie sono spesso imprescindibili per rispettare i tempi stimati del progetto.

Nel prossimo capitolo vedremo come utilizzare gli strumenti offerti da Visual Studio per ottimizzare ulteriormente il codice JavaScript.

20
Gli strumenti di Visual Studio per lo sviluppo web

Nel precedente capitolo abbiamo introdotto alcune librerie JavaScript che migliorano l'esperienza di sviluppo di applicazioni web ibride fornendo componenti già pronti per l'uso che sfruttano sia HTML che JavaScript. In questo capitolo vediamo come sfruttare alcuni strumenti per ottimizzarne le performance e per pre-processare alcuni file.

Quando scriviamo codice, prima che questo venga eseguito, la macchina deve compilarlo in codice nativo. Quando sviluppiamo per il Web, possiamo traslare questa affermazione dicendo che quando creiamo file CSS o JavaScript, dobbiamo precompilarli prima di distribuirli via web. Esistono diversi casi in cui quanto appena detto corrisponde a realtà.

Quando referenziamo un file JavaScript all'interno di una pagina web, questo viene inviato al client così com'è. Se una pagina referenzia più file JavaScript il client esegue una richiesta per ogni file. Queste affermazioni portano alla considerazione che avere file molto grandi o referenziare troppi file può causare problemi di performance di rete della nostra applicazione. Questa stessa affermazione si applica non solo ai file JavaScript, ma anche ai file CSS. Nel corso degli anni si è ricorso a diverse tecniche per mitigare il problema e si è arrivati a due soluzioni complementari: minification, cioè la capacità di ridurre le dimensioni di un file, e bundling, cioè la capacità di unire più file in un solo file. Entrambe le tecniche prevedono che i file JavaScript e CSS subiscano una sorta di precompilazione prima di essere usate.

Per fare un altro esempio, nei file CSS molto spesso ci sono tante informazioni ripetute come i colori, le dimensioni, i margini e così via. Quando dobbiamo modificare uno di questi valori, per esempio un colore, dobbiamo scorrere i file CSS e modificare il valore ovunque. È chiaro che un'organizzazione del genere comporta grosse difficoltà di manutenzione con alte probabilità di errore. Per questo motivo sono state inventate sintassi simil-CSS (come SCSS) che vengono pre-processate e che producono in output file CSS da inviare ai client.

Questi esempi rendono chiaro il perché non sia sempre una buona idea sviluppare e inviare direttamente il nostro sorgente al client, ma sia molto meglio preprocessarlo per avere un ventaglio di opportunità molto più ampio.

Nel corso del capitolo parleremo delle tecniche appena menzionate e di come metterle in pratica sfruttando alcuni strumenti di terze parti integrati in Visual Studio (Gulp, Grunt, e NPM).

Bundling e minification

Queste due tecniche permettono di ottimizzare in modo importante le prestazioni di rete della nostra applicazione. Molte applicazioni di tuning delle applicazioni web considerano la mancanza di bundling e minification un problema e lo segnalano.

> *Due esempi di applicazioni di tuning dei siti web sono PageSpeed, sviluppato da Google e disponibile all'indirizzo http://aspit.co/bph, e YSlow disponibile all'indirizzo http://aspit.co/bpj.*

Inoltre, i motori di ricerca penalizzano siti (soprattutto mobile) che non sfruttano queste tecniche. Il risultato è che il rank di questi siti è più basso e quindi appaiono più in basso nelle ricerche.

Minification

La minification è la tecnica che prende in input un file e ne riduce le dimensioni applicando una serie di regole. Alcune di queste regole prevedono la modifica del nome, al fine di renderlo più corto, delle variabili interne e dei parametri dei metodi, la rimozione di commenti, spazi e altri caratteri inutili e altro ancora. Il risultato è un file poco comprensibile ma di dimensioni notevolmente ridotte. La percentuale di riduzione del codice dipende molto dai nomi originali, ma si stima che si possano ridurre le dimensioni originali fino al 60%. Prendiamo come esempio il seguente codice JavaScript che mostra il codice originale e il codice minificato.

Esempio 20.1

```
//codice originale
var calculateArea = function(xSize, ySize) {
   //multiply input parameters
   return xSize * ySize;
}

//codice minificato
var calculateArea=function(a,b){return a*b};
```

Il codice originale è di 103 caratteri mentre quello minificato ne conta 44 con un'ottimizzazione del 55%. In un file di così piccole dimensioni la differenza è risibile, ma al crescere delle dimensioni del file il guadagno è notevole. Grazie a questa tecnica, sulla rete viaggiano molti meno dati e quindi la nostra applicazione è molto più veloce da caricare.

Lo stesso discorso si può applicare non solo al codice JavaScript ma anche ai file CSS, come nel prossimo esempio.

Esempio 20.2

```
//codice originale
```

```
.myStyle {
   border-width: 1px;
   border-style: solid;
   border-color: black;
   background-color: red;
}

.otherstyle{
   color: #000;
   white-space: nowrap;
   font-size: 10px
}

//codice minificato
.myStyle{border-width:1px;border-style:solid;border-color:#000;
background-color:red}.otherstyle{color:#000;white-space:nowrap;
font-size:10px}
```

In questo caso, il codice originale è di 109 caratteri mentre quello minificato è di 85 caratteri, con un risparmio di circa il 25%. Anche in questo caso, con file di così piccole dimensioni, il vantaggio è minimo ma, all'aumentare delle dimensioni del file originale, il risparmio diventa più consistente e si hanno maggiori benefici. Se sommiamo il risparmio che possiamo ottenere minificando sia CSS sia JavaScript, appare evidente che il miglioramento delle performance di rete è veramente importante. Passiamo ora a vedere il bundling, che offre ulteriori vantaggi.

Bundling

Il bundling è la tecnica che prevede l'unione di più file in un unico file, al fine di ridurre le richieste di rete che il client fa verso il server. Se questi file sono anche minificati, otteniamo un unico file minificato e abbiamo quindi limitato al massimo i dati da scaricare.

Creare un solo unico file JavaScript e un solo unico file CSS può non essere la soluzione migliore. Se da un lato riduciamo al minimo il numero di file scaricati e la loro dimensione, dall'altro non sfruttiamo le capacità di parallelismo dei browser. Infatti, i browser sono in grado di scaricare più file contemporaneamente (il numero di file contemporanei varia da browser a browser) e questo significa che avere pochi file di dimensioni ridotte potrebbe essere più performante di avere un solo file di grandi dimensioni. Come sempre, occorre fare dei test prima di scegliere una strada o l'altra.

È chiaro che l'insieme di bundling e minification rappresenta una scelta praticamente obbligata per creare applicazioni che siano performanti dal punto di vista della rete. Tuttavia, mentre stiamo sviluppando quest'ottimizzazione, è inutile, anzi addirittura dannosa nei casi in cui dobbiamo eseguire il debug dei file JavaScript. In questi casi può venire in aiuto ASP.NET Core.

Integrazione con ASP.NET Core

Nel Capitolo 6 abbiamo introdotto il tag helper `environment`, che permette di renderizzare frammenti di HTML in base all'ambiente in cui ci troviamo. Nel nostro caso usiamo questo tag helper per renderizzare dei tag `link` o `script` differenti a seconda dell'ambiente. Nel caso in cui ci troviamo nell'ambiente di sviluppo, utilizziamo i codici sorgenti; se, invece, ci troviamo nell'ambiente di produzione, utilizziamo i file sottoposti a bundling e minification come viene mostrato nell'esempio.

Esempio 20.3

```
<environment include="Development">
   <link rel="stylesheet" href="~/css/bootstrap.css" />
   <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
   <link rel="stylesheet" href="~/dist/bundle.min.css" />
</environment>
```

In questo esempio abbiamo utilizzato il tg `link` per referenziare file CSS ma, allo stesso modo, possiamo usare il tag `script` per referenziare file JavaScript. Entrambi i tag sono in realtà anche dei tag helper, il cui utilizzo è spiegato nella prossima sezione.

Link e Script tag helper

Nell'Esempio 20.3 abbiamo visto che il tag `link` nell'ambiente di sviluppo punta al file di Bootstrap e a un altro file CSS, mentre negli altri ambienti punta a un unico file che contiene entrambi i file in bundling. Tuttavia, in un'applicazione reale, la referenza al file di Bootstrap non dovrebbe puntare a un file locale, bensì a un file servito da una CDN, così da ottimizzare al massimo le performance, in quanto molto probabilmente il file si trova già nella cache del browser.

Essendo il file fornito da una CDN su cui non abbiamo controllo, dobbiamo anche prevedere un meccanismo di fallback che fornisca il file dal server locale qualora la CDN non dovesse essere in grado di fornirlo. In questi casi entra in gioco il tag helper `link`, che offre una

serie di proprietà che eseguono un test per capire se il file è stato scaricato dalla CDN e specificano da quale sorgente alternativa scaricare il file nel caso il test fallisca. Le proprietà sono:

- `asp-fallback-test-class`: specifica una classe CSS che si trova nel file che intendiamo scaricare dlla CDN;
- `asp-fallback-test-property`: specifica una proprietà della classe CSS che si deve trovare nella classe CSS specificata in precedenza;
- `asp-fallback-test-value`: specifica il valore che deve avere la proprietà specificata in precedenza;
- `asp-fallback-href`: specifica l'url del file da scaricare nel caso il test fallisca.

L'Esempio 20.4 mostra come scaricare il file CSS di Bootstrap da una CDN, come specificare il test (verificando che esista una classe CSS `sr-only` con la proprietà `position` impostata su `absolute`) e come fornire un fallback dal server locale.

Esempio 20.4

```
<link rel="stylesheet" href="
   https://ajax.aspnetcdn.com/ajax/bootstrap/4.1.1/css/bootstrap.min.css"
   asp-fallback-href="~/css/bootstrap.min.css"
   asp-fallback-test-class="sr-only" asp-fallback-test-property="position"
  asp-fallback-test-value="absolute" />
```

Il tag helper `link` espone anche la proprietà booleana `asp-append-version` tramite la quale indichiamo ad ASP.NET Core di includere un hash del file nell'url, nel caso il file sia servito da locale. Questa funzionalità torna utile quando dobbiamo servire un file da locale e cambiamo il contenuto del file. Se il file si trova già nella cache del browser e non includiamo l'hash, il browser utilizzerà la versione in cache e ignorerà la nuova versione. Se invece mettiamo l'hash nell'url, il server invierà al client un url diverso da quello precedente e quindi il browser sarà forzato a eseguire il download del nuovo file.

Il tag `script` non è molto diverso dal tag `link` in quanto svolge gli stessi compiti, ma con una differente sintassi dovuta al fatto che il test per capire se il file JavaScript è stato scaricato dalla CDN non è come il test per i CSS. Invece delle tre proprietà di test per i CSS, abbiamo solamente la proprietà `asp-fallback-test`, che contiene un'espressione JavaScript che deve essere vera per confermare il download del file dalla CDN. Nel caso l'espressione sia falsa, si procede al download dal fallback espresso dalla proprietà `asp-fallback-href`.

Esempio 20.5

```
<script
   src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
   asp-fallback-src="~/scripts/bootstrap.min.js"
   asp-fallback-test="window.jQuery.fn.modal">
</script>
```

L'integrazione di bundling, minification e CDN con ASP.NET Core è semplice grazie ai tag `environment`, in combinazione con `script` e `link`. Tuttavia non abbiamo ancora visto come applicare minification e bundling ai file sorgenti all'interno di Visual Studio. Questi task possono essere eseguiti attraverso due strumenti, Gulp e Grunt, che hanno in comune il fatto di dover essere scaricati da NPM, che è l'argomento della prossima sezione.

Utilizzare NPM

NPM è l'acronimo di NodeJS Package Manager ed è un repository di package JavaScript oramai diffusissimo tra tutti gli sviluppatori client. All'interno di NPM ci sono package di librerie JavaScript così come anche strumenti (detti anche plugin) di sviluppo, come Gulp e Grunt.

> *Microsoft consiglia di utilizzare NPM esclusivamente per scaricare plugin di sviluppo e non librerie (in quanto queste possono essere scaricate tramite LibMan). Tuttavia questa è solo una raccomandazione e non ci sono problemi a scaricare anche librerie da NPM.*

NPM si abilita all'interno del progetto creando il file `package.json` e salvandolo nella root del progetto. Visual Studio semplifica questo compito, offrendo un template nella finestra per creare un nuovo file come mostra la Figura 20.1.

Il file creato da Visual Studio prevede le proprietà minime per il funzionamento di NPM (`name`, `version`, `private`), più la proprietà `devDependencies` che è la più importante per i nostri scopi. Questa proprietà, inizialmente vuota, contiene il dictionary all'interno del

quale specifichiamo i plugin che vogliamo scaricare: la chiave rappresenta il nome del plugin, mentre il valore rappresenta la versione. Ogni volta che salviamo il file, Visual Studio si connette a NPM e scarica i componenti lanciando il comando `npm install`.

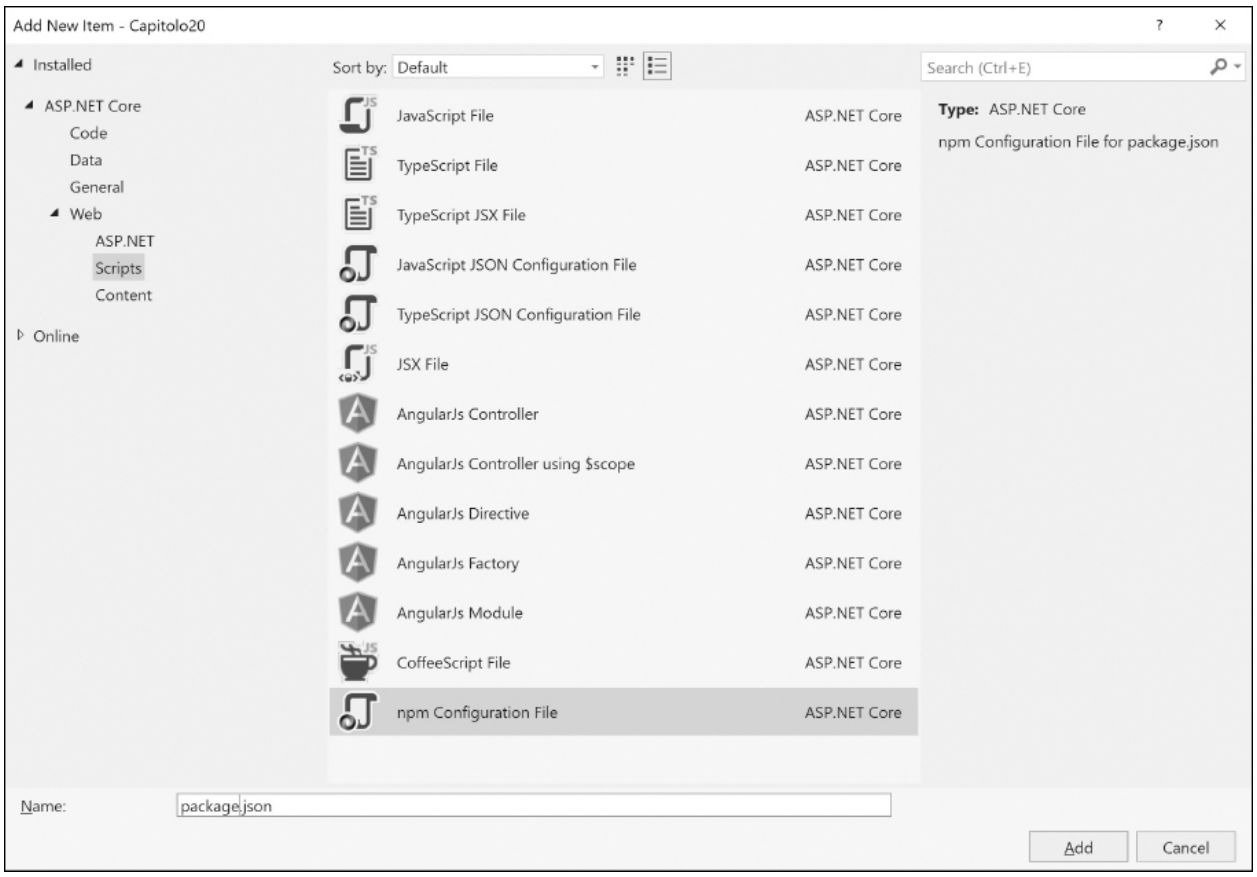


Figura 20.1 – La finestra di Visual Studio che permette di creare il file `package.json`.

La versione può essere espressa in tre modi:

- numero di versione: scarica l'esatta versione;
- numero di versione preceduto da ^: scarica l'ultima major version;
- numero di versione preceduto da ~: scarica l'ultima minor version;

L'Esempio 20.6 mostra il file `package.json` con le diverse opzioni per specificare la versione.

Esempio 20.6

```
{
   "version": "1.0.0",
   "name": "asp.net",
   "private": true,
      "devDependencies": {
         "grunt": "^1.0.2",
         "gulp": "~3.9.1",
         "gulp-minify": "2.1.0",
      }
}
```

Oltre alla proprietà `devDependencies` esiste la proprietà `dependencies`, che viene usata per scaricare librerie da usare nel codice, invece che plugin usati a design time, e che ha la stessa struttura di `devDependencies` all'interno del file `package.json`. Il motivo per cui esistono due sezioni è per organizzare al meglio il file `package.json` e per facilitare il deploy in produzione, in quanto lanciando il comando `npm install --only=production`, si scaricano solo le librerie referenziate nella sezione `dependencies`.

NPM è uno strumento estremamente potente che tramite linea di comando offre molte altre opzioni. Tuttavia non dobbiamo preoccuparci di conoscere il comando e le sue opzioni, in quanto l'integrazione con Visual Studio esegue tutto al posto nostro nel momento in cui salviamo il file.

Per maggiori informazioni su NPM è disponibile la documentazione sul sito ufficiale all'indirizzo http://aspit.co/bov.

Ora che sappiamo come recuperare i plugin di sviluppo necessari a processare i nostri file JavaScript e CSS, vediamo come utilizzarli partendo da Gulp.

Utilizzare Gulp

Gulp è un motore (detto task runner) che esegue una dietro l'altra funzioni JavaScript (dette task) che hanno un input e ritornano un output che andrà in input alla funzione successiva. Gulp non è a conoscenza di cosa facciano i task e di quale sia il loro input e il loro output, in quanto queste specifiche appartengono ai task. Gulp è solo responsabile dell'invocazione dei task e del trasporto dei dati che questi ricevono e ritornano.

> *Microsoft non è proprietaria di Gulp: questo strumento è open source su github ed è stato solamente integrato in Visual Studio.*

La definizione della catena di task è fatta nel file `gulpfile.js` situato nella root del progetto. In testa al file importiamo le dipendenze del nostro script e, successivamente, specifichiamo i vari task da eseguire.

Per importare una dipendenza, la dobbiamo scaricare da NPM e utilizzare il metodo `require` con il nome della dipendenza. Visto che il nostro scopo è quello di eseguire bundling e minification, dobbiamo scaricare i plugin che eseguono queste operazioni tramite NPM e referenziarli nel file `gulpfile.json` per orchestrarli con Gulp. I plugin sono `gulp-concat` (che concatena più file), `gulp-minify` (che minifica i file JavaScript) e `gulp-clean-css` (che minifica i file CSS). L'Esempio 20.7 mostra come importare queste dipendenze.

Esempio 20.7

```
const gulp = require('gulp');
const concat = require('gulp-concat');
const cleancss = require('gulp-clean-css');
const minify = require('gulp-minify');
```

Per creare un task, utilizziamo il metodo `task` della classe `gulp`, alla quale passiamo il nome del task e una funzione all'interno della quale specifichiamo gli step necessari a completare il task. Ogni step prende in input l'output dello step precedente, detto stream, e ritorna a sua volta uno stream che viene passato allo step successivo. Arrivati all'ultimo step, il task si conclude.

Generalmente, il primo step è il metodo `src` della classe `gulp`, al quale passiamo in input un array di file. Questi file vengono inseriti in uno stream che passiamo allo step successivo, invocato tramite il metodo `pipe`, sempre della classe `gulp`.

`pipe` accetta in input il metodo che lavora lo stream proveniente dallo step precedente e che ritorna un nuovo stream. Questo viene preso da `pipe` che lo passa allo step successivo invocato di nuovo tramite `pipe`, creando così un processo che si itera fino all'ultima chiamata.

Nell'Esempio 20.8 creiamo il task `bundle-js` per lavorare i file JavaScript. In questo esempio sfruttiamo il metodo `src` per recuperare i file JavaScript da lavorare, il metodo `minify` per eseguirne la minification, il metodo `concat` per eseguire il bundling (specificando il nome del file di bundle) e, infine, il metodo `dest` della classe `gulp` per salvare il file nel percorso specificato.

Nello stesso esempio creiamo anche il task `bundle-css`, per lavorare i file CSS, che sfrutta lo stesso pattern visto in precedenza: selezioniamo i file con il metodo `src`, ne eseguiamo la minification con il metodo `cleancss`, ne eseguiamo il bundling con il metodo `concat` e, infine, salviamo il file con il metodo `dest`.

Esempio 20.8

```
gulp.task('bundle-js', function () {
   return gulp
      .src(['./wwwroot/js/site.js', './wwwroot/js/site2.js'])
      .pipe(minify())
      .pipe(concat('bundle.min.js'))
      .pipe(gulp.dest('./wwwroot/dist'));
});

gulp.task('bundle-css', function () {
   return gulp
      .src(['./wwwroot/css/site.css', './wwwroot/css/site2.css'])
      .pipe(cleancss())
      .pipe(concat('bundle.min.css'))
      .pipe(gulp.dest('./wwwroot/dist'));
});
```

Una volta salvato il file, dobbiamo eseguirne i task tramite la CLI di Gulp. Tuttavia l'integrazione di Gulp all'interno di Visual Studio permette di lanciare i task sfruttando una UI (visibile nella Figura 20.2) accessibile tramite il menù View - Other Windows - Task Runner Explorer.

La maschera mostra sulla sinistra i task nel file `gulpfile.js`. Con un doppio click del mouse su uno dei task, eseguiamo il task e, sulla destra, la finestra ne mostra il risultato. Sulla destra c'è anche il tab `bindings`, tramite il quale possiamo impostare l'esecuzione di un task prima o dopo la compilazione, all'apertura del progetto o nella fase di clean del progetto.

Per maggiori informazioni su Gulp, è disponibile la documentazione sul sito ufficiale, all'indirizzo: http://aspit.co/bow.

Come abbiamo detto, Gulp non è il solo task runner in Visual Studio, in quanto c'è anche Grunt, che tratteremo nella prossima sezione.

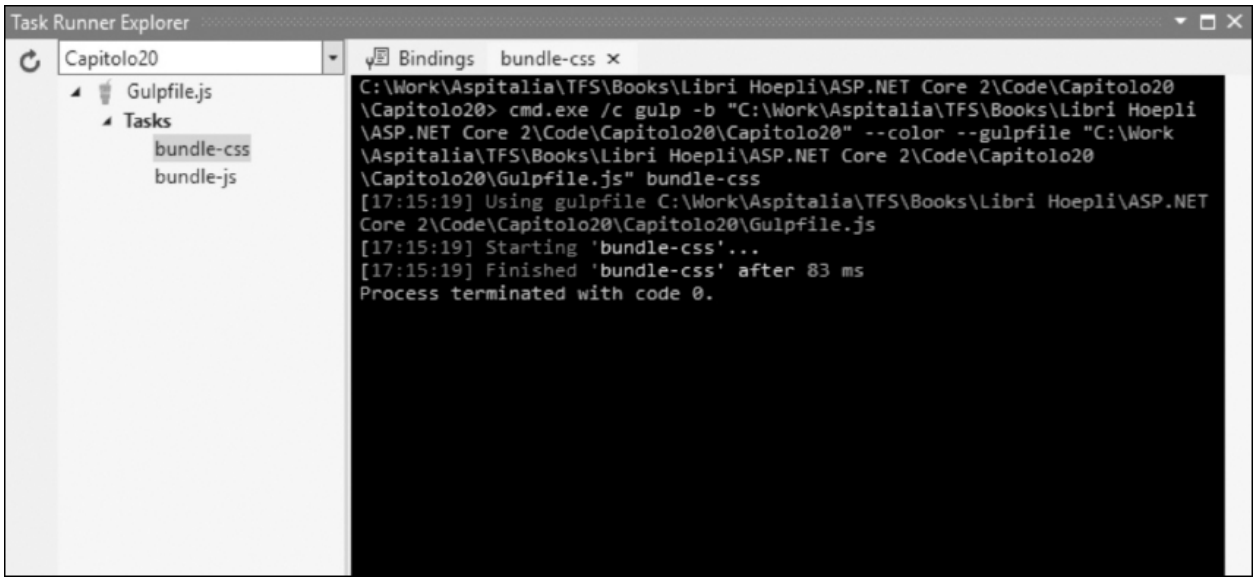


Figura 20.2 – La finestra di Visual Studio che permette di lanciare i task di Gulp.

Utilizzare Grunt

Grunt è un altro task runner integrato in Visual Studio, che svolge le stesse funzioni di Gulp ma in modo leggermente diverso e sfruttando altri plugin. La scelta tra Gulp e Grunt è dettata dalle conoscenze e dai gusti personali ma c'è da sottolineare che mentre Gulp è costantemente manutenuto ed evoluto, Grunt dopo la release 1.0 ha avuto solo un paio di minor release per bug fixing.

Prima di passare a vedere come funziona Grunt, dobbiamo specificare che i plugin di Grunt e Gulp non sono compatibili e quindi, per poter eseguire minification e bundling, dobbiamo scaricarne di nuovi da NPM. Questi nuovi plugin sono `Grunt` per scaricare Grunt, `grunt-contrib-uglify` per lavorare i file JavaScript e `grunt-contrib-cssmin` per lavorare i file CSS.

Per abilitare Grunt in Visual Studio, dobbiamo creare il file `gruntfile.js` nella root del progetto e modificarlo per aggiungere i task da eseguire. Poiché Grunt viene eseguito da NodeJS, il file deve dichiarare innanzitutto l'entry point, impostando la variabile `module.exports` con una funzione all'interno della quale impostiamo i task da eseguire.

Esempio 20.9

```
module.exports = function (grunt) {
   // Configurazione grunt
}
```

La funzione che agisce come entry point accetta in input un'istanza della classe tramite la quale configuriamo Grunt. Il primo metodo che utilizziamo è `initConfig`, al quale dobbiamo passare un oggetto JSON che contiene i dati di configurazione dei plugin e anche altro. La prima proprietà dell'oggetto JSON è `pkg`, che facciamo puntare al file `package.json`, così che Grunt possa assicurarsi di usare la versione corretta dei plugin, scaricandoli automaticamente tramite NPM, se necessario. Successivamente creiamo una proprietà per ogni plugin che utilizziamo, dove il nome della proprietà deve corrispondere al nome del plugin e il valore assegnato alla proprietà è un oggetto che configura il plugin. Quest'oggetto è specifico per ogni plugin quindi dobbiamo consultarne la documentazione per capire come configurarlo. Ogni singolo plugin configurato corrisponde a un task che possiamo invocare.

Nel prossimo esempio vediamo come creare l'oggetto JSON per eseguire bundling e minification di JavaScript e CSS attraverso i plugin specificati in precedenza.

Esempio 20.10

```
grunt.initConfig({
   pkg: grunt.file.readJSON('package.json'),
   uglify: {
      build: {
         files: {
            './wwwroot/dist/bundle.min.js': [
               './wwwroot/js/site.js',
                  './wwwroot/js/site2.js']
         }
      }
   },
   cssmin: {
      build: {
         files: {
            'wwwroot/dist/bundle.min.css': [
               'wwwroot/css/site.css',
                  'wwwroot/css/site2.css']
         }
      }
   }
});
```

Il plugin `grunt-contrib-uglify` ha come nome `uglify` e l'oggetto in input specifica i file JavaScript da processare e il nome del file di bundle. Il plugin `grunt-contrib-cssmin` ha come nome `cssmin` e la medesima configurazione di `uglify` con la sola differenza di processare i file CSS. Una volta configurati i plugin, dobbiamo utilizzare il metodo `loadNpmTask` dell'istanza di Grunt per importare i plugin. L'utilizzo di questo metodo è mostrato nell'Esempio 20.11.

Esempio 20.11

```
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-cssmin');
```

L'ultimo metodo dell'oggetto di Grunt da invocare è `registerTask`, che crea task combinando quelli definiti nel JSON passato al metodo `initConfig`.

`registerTask` prende in input il nome del task e un array di stringhe che corrispondono ai nomi dei plugin. Un utilizzo è mostrato nel prossimo esempio.

Esempio 20.12

```
grunt.registerTask('build', ['uglify', 'cssmin']);
```

L'ultimo step consiste nel lanciare i task da Visual Studio attraverso la finestra Task Runner Explorer, esattamente come abbiamo visto per Gulp. L'unica variante consiste nel fatto che la finestra mostra un task per ogni plugin configurato più i task configurati con il metodo `registerTask` sotto il nodo Alias Task come mostra la Figura 20.3.

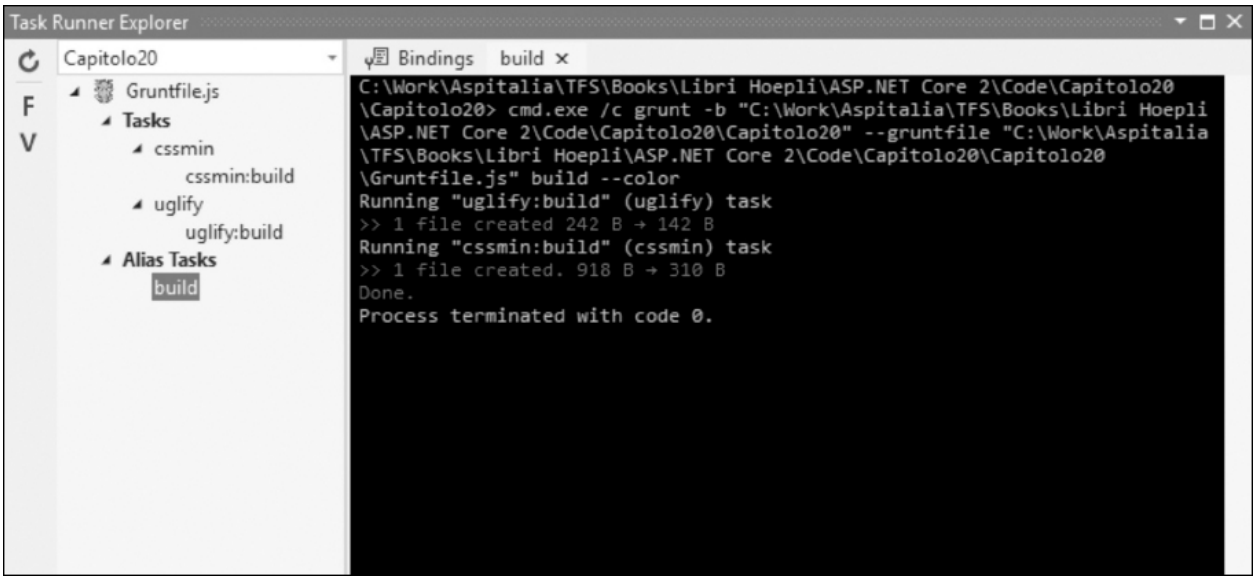


Figura 20.3 – La finestra di Visual Studio che permette di lanciare i task di Grunt.

Per maggiori informazioni su Grunt, è disponibile la documentazione sul sito ufficiale, all'indirizzo: http://aspit.co/box.

Conclusioni

In questo capitolo abbiamo fatto la conoscenza di due tecniche molto importanti per le performance di rete di un'applicazione: minification e bundling. Oltre a queste tecniche, possiamo anche abilitare la compressione GZip tramite un middleware o sul reverse proxy, con lo scopo di ottimizzare ulteriormente il traffico dati.

Inoltre, abbiamo visto come Visual Studio non offra solo strumenti dedicati a chi sviluppa funzionalità lato server ma anche funzionalità per gli sviluppatori che lavorano prettamente sulla UI e che necessitano maggiormente di queste funzionalità.

Grazie all'integrazione di Visual Studio con strumenti come NPM, Gulp, Grunt e, implicitamente, NodeJS, possiamo pre-processare qualunque tipo di file prima di utilizzarlo nelle nostre pagine e risparmiare così lavoro manuale a design time, ottimizzando le performance a run time.

Ora cambiamo argomento e parliamo di come creare Single Page Application in Visual Studio con ASP.NET Core e Angular.

21
Single Page Application e ASP.NET Core

Negli ultimi due capitoli abbiamo introdotto i principali framework, librerie e strumenti per lo sviluppo lato client di applicazioni ibride. Abbiamo visto, inoltre, come Visual Studio permetta facilmente l'integrazione di questi strumenti all'interno delle nostre applicazioni, così da rendere lo sviluppo più semplice rispetto al passato.

In questo capitolo vediamo come Visual Studio permetta lo sviluppo non solo di applicazioni ibride ma anche di Single Page Application (SPA d'ora in poi). Le SPA sono applicazioni che girano interamente sul browser e che sfruttano il server solamente per recuperare i file JavaScript, HTML e CSS e per leggere e scrivere dati. Questo modello di sviluppo presenta indubbi vantaggi quando si tratta di performance e usabilità ma ha come controindicazione un utilizzo massiccio di codice JavaScript sia dal punto di vista infrastrutturale sia dal punto di vista del codice di business.

Per ridurre al minimo il codice infrastrutturale necessario a creare una SPA, sono nati negli anni diversi framework e librerie, come Angular, ReactJS, Vue.JS e Aurelia. Poiché i primi due sono attualmente i più diffusi, Microsoft ha deciso di integrarli in Visual Studio, offrendo un template già pronto e l'integrazione con strumenti di sviluppo come, per esempio, NPM. Nel corso del capitolo parleremo sia di Angular sia di ReactJS, ma prima di iniziare ad affrontarli, approfondiamo il concetto di SPA.

Introduzione alle Single Page Application

Come abbiamo detto nell'introduzione del capitolo, una SPA è un'applicazione che gira interamente sul client e che sfrutta il server solo per recuperare i file JavaScript, CSS e HTML e per recuperare i dati. Questo significa che quando un utente digita l'url dell'applicazione, il server risponde con tutti i file necessari affinché l'applicazione possa essere renderizzata sul browser e possa essere navigata senza ricorrere a ulteriori richieste al server, se non per recuperare e persistere i dati visualizzati nelle pagine effettuando chiamate AJAX a delle WebAPI.

Questo significa che se la nostra applicazione è composta da 20 pagine, alla prima richiesta il server invia una pagina HTML (detta master) contenente i riferimenti a tutti i file JavaScript e CSS necessari a renderizzare le 20 pagine. Il codice HTML delle 20 pagine è già incluso nei file JavaScript, velocizzando così il processo di renderizzazione.

Quando il browser ha finito di scaricare i file necessari, il codice JavaScript parte e identifica quale sia la home della nostra applicazione, ne prende il contenuto HTML e lo attacca al tag body della pagina master. Quando l'utente clicca su un url o compie un'altra azione che comporta la navigazione verso un'altra pagina, il codice JavaScript intercetta questa navigazione, identifica quale sia la pagina verso cui stiamo navigando, ne recupera il contenuto HTML, svuota il tag body della pagina master e lo riempie con il contenuto della nuova pagina. Questo processo di navigazione non comporta un post della pagina al server o una nuova richiesta al server, in quanto viene tutto completamente gestito lato client.

> *Da questa spiegazione emerge che dal server scarichiamo una sola pagina HTML. Questo è il motivo per cui questo genere di applicazioni viene chiamato Single Page Application.*

Ogni pagina che viene caricata nel master ha del codice JavaScript che la gestisce e con il quale può essere messa in binding. Questo codice è anche responsabile di invocare via AJAX il server qualora la pagina necessiti di visualizzare dati per essere renderizzata correttamente o necessiti di persistere dati. La Figura 21.1 mostra il funzionamento di una SPA.

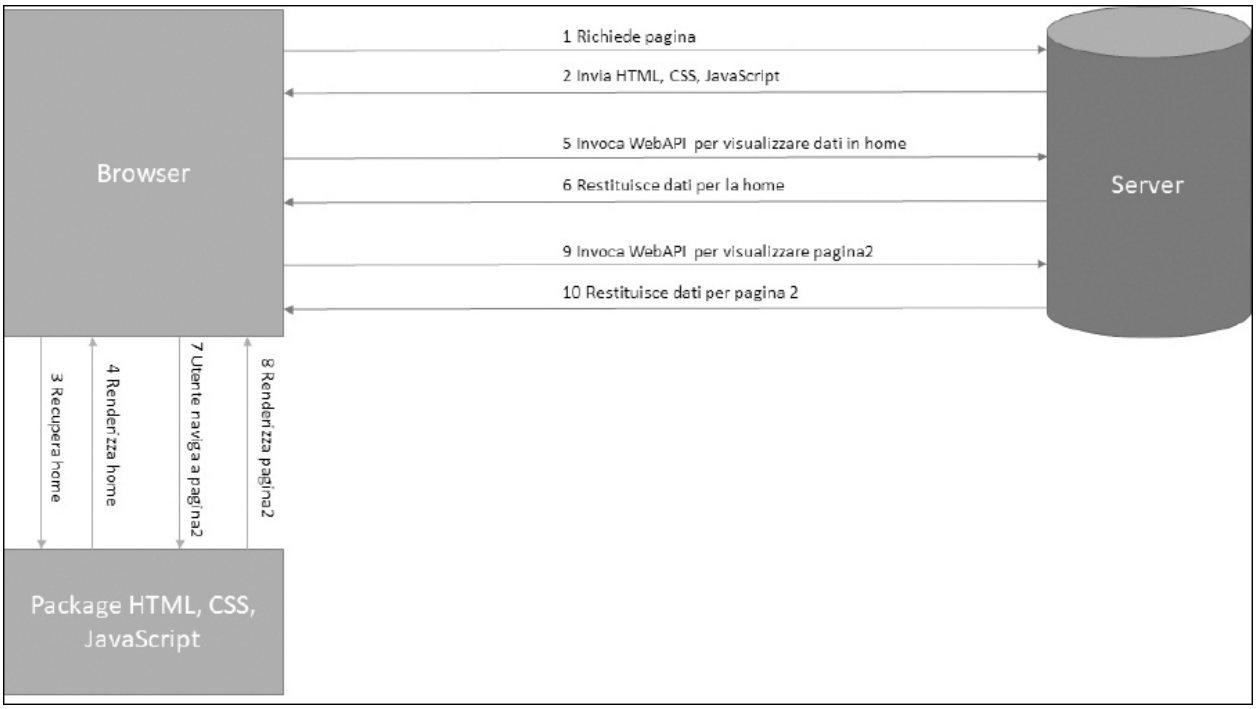


Figura 21.1 – L'interazione tra client e server avviene solo per la richiesta iniziale e per recuperare i dati per le pagine. Il resto avviene in locale.

Da questa breve introduzione emergono alcune importanti considerazioni. La prima è che le performance lato client delle SPA sono notevolmente superiori rispetto a quelle delle applicazioni ibride, poiché la navigazione avviene lato client senza alcun contatto col server e quindi senza alcuna latenza.

La seconda considerazione è che anche il server beneficia di migliori performance. Una volta inviati al client i file necessari, l'unica interazione che questo ha con il client è per lo scambio di dati via WebAPI. Questo significa che ci sono meno richieste al server e che queste richieste riguardano, nella maggior parte dei casi, solo lo scambio di dati. Rispetto alle applicazioni ibride che prevedono un continuo pattern di richiesta/risposta con il server con scambio di codice HTML, la quantità di dati che viaggia sulla rete è minore.

La terza considerazione è che anche l'usabilità delle SPA è notevolmente superiore a quella delle applicazioni ibride. Poiché tutto è gestito lato client, l'utente non ha l'effetto di ricaricamento della pagina a ogni richiesta e possiamo anche usare animazioni per passare da una pagina all'altra. A tutti gli effetti, all'utente sembra di utilizzare un'applicazione nativa piuttosto che un'applicazione web.

Come sempre, ci sono pro e contro nello sviluppare una SPA rispetto a un'applicazione ibrida. Come si può immaginare, il codice JavaScript per gestire la parte infrastrutturale (intercettazione degli eventi di navigazione, rendering delle pagine nella master, caricamento dei file JavaScript su richiesta e non allo startup e altro ancora) è piuttosto corposo e complesso così come può esserlo anche quello di business. Nonostante i notevoli miglioramenti apportati al linguaggio negli ultimi anni, JavaScript non è ancora un linguaggio che si sposa molto bene in progetti di una certa entità e richiede molta disciplina agli sviluppatori. Gli strumenti di sviluppo sono notevolmente migliorati ma non siamo ancora ai livelli degli strumenti presenti per linguaggi come C#, e quindi, anche sotto questo punto di vista, ci troviamo svantaggiati nello sviluppare SPA.

Come sempre, la scelta tra un modello di sviluppo SPA e un modello ibrido dipende dalle conoscenze tecniche del team, dalle richieste del cliente e dai costi e dalle tempistiche. Se in un team ci sono molti sviluppatori JavaScript, probabilmente una SPA è la soluzione migliore, mentre se si hanno più sviluppatori C#, la scelta di creare un'applicazione ibrida è probabilmente la strada più sicura.

Se si decide di creare SPA, non si può prescindere dall'utilizzo di un framework che ne semplifichi lo sviluppo. Come abbiamo detto nell'introduzione, negli anni sono nati diversi framework che si occupano di tutti gli aspetti infrastrutturali, lasciando a noi il solo compito di configurare l'applicazione e scrivere il codice di business. Grazie a questi framework, il codice necessario a creare una SPA è notevolmente ridotto rispetto a quanto dovremmo scriverne se non li utilizzassimo. Il primo framework che prendiamo in considerazione è Angular.

Angular

Angular è il framework prodotto da Google per creare SPA. Angular è il successore di AngularJS, con cui però condivide parte della filosofia e poco altro. Angular è stato riscritto da zero utilizzando TypeScript come linguaggio di programmazione, con un'architettura modulare che permette di componentizzare la nostra applicazione in modo semplice.

Oltre al framework, Angular gode anche di un ecosistema che utilizza strumenti già consolidati nello sviluppo web (come NPM e WebPack) e strumenti ad-hoc come Angular-CLI. Grazie a questo ecosistema, lo sviluppo di una SPA basata su Angular è meno complesso rispetto a uno sviluppo effettuato con il suo predecessore.

Visual Studio offre un template che genera un'applicazione che sfrutta ASP.NET Core per le WebAPI e Angular per la UI attraverso il wizard di creazione di un'applicazione web, come è mostrato nella Figura 21.2. Al momento della stesura di questo libro, il template genera un'applicazione Angular 5 ma in un aggiornamento futuro di Visual Studio il template verrà aggiornato per generare un'applicazione sfruttando Angular 6.

Il progetto creato da Visual Studio genera lo scheletro di un'applicazione ASP.NET Core con un controller per le WebAPI che fornisce dati preimpostati e con la cartella ClientApp che contiene l'applicazione Angular, che è quella che ci interessa.

Per creare lo scheletro di quest'ultima, Visual Studio sfrutta Angular-CLI. Questo strumento, che fa parte del SDK di Angular ed è scaricato da NPM, permette di creare un'applicazione da zero e di aggiungervi componenti man mano che si sviluppa l'applicazione. Permette inoltre di compilare la nostra applicazione sia in debug sia in produzione, avviare un web server per il debug e altro ancora.

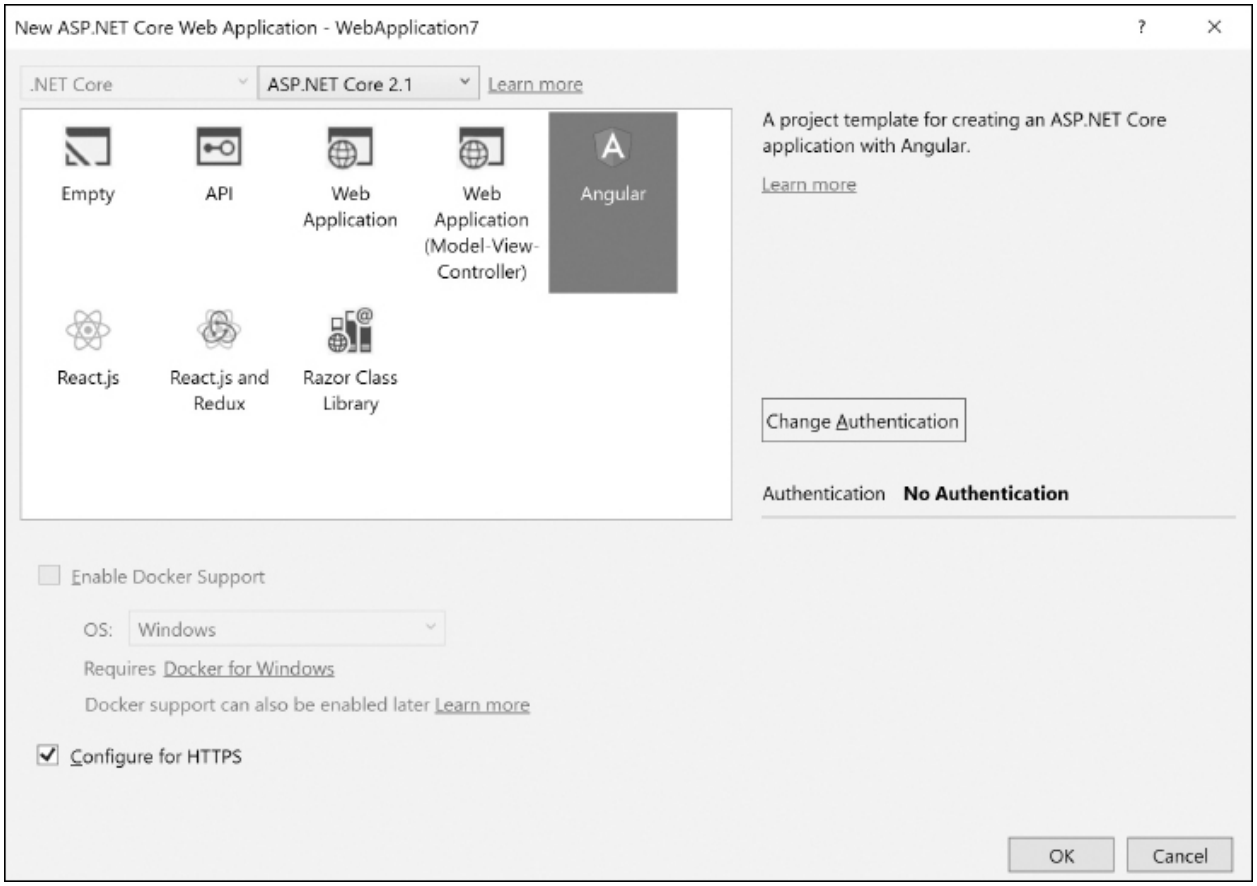


Figura 21.2 – Il wizard di creazione di un'applicazione web include l'opzione per Angular.

Accenneremo ad alcuni comandi della CLI nel corso del capitolo ma, per maggiori informazioni, si può consultare il sito di Angular-CLI, all'indirizzo: http://aspit.co/bpn.

Una volta terminata la creazione del progetto, la finestra Solution Explorer appare come nella Figura 21.3.

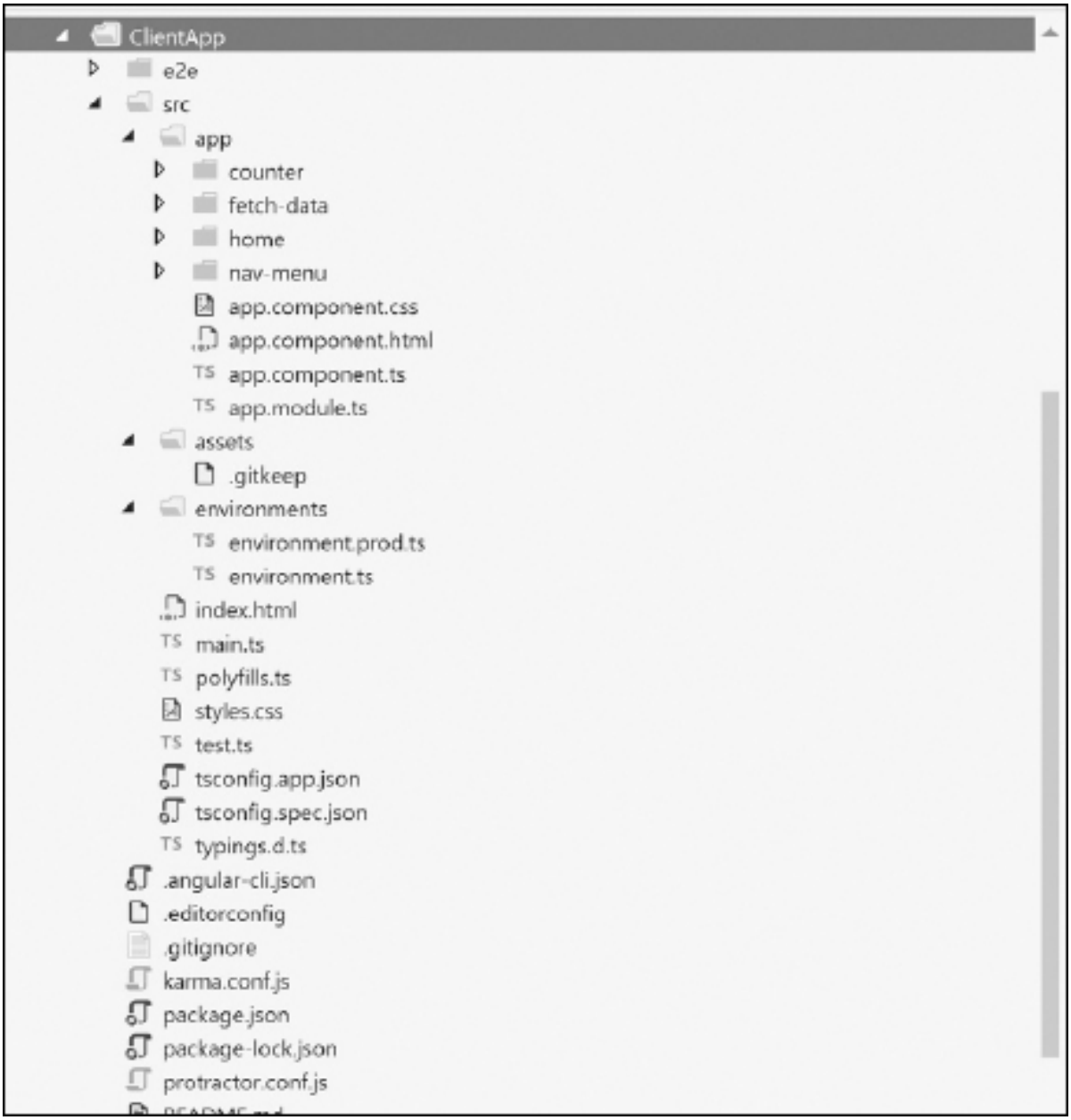


Figura 21.3 – La finestra Solution Explorer mostra il codice di un'applicazione Angular appena creata tramite Visual Studio.

Nella root della cartella ClientApp troviamo i file necessari a configurare l'ambiente come il file `tsconfig.json`, che stabilisce le regole di compilazione di TypeScript, il file `tslint.json`, che specifica le regole di formattazione del codice, il file `package.json`, per stabilire strumenti e librerie da scaricare da NPM (tra cui quelle di Angular) e il file `.angular-cli.json`, per configurare Angular-CLI.

Il codice sorgente si trova nella cartella src, all'interno della quale troviamo il file `index.html`, che sarà la pagina HTML master, il file `main.ts`, che è responsabile dell'inizializzazione dell'applicazione, il file `styles.css`, che contiene gli stili CSS e il file `polyfill.ts`, che contiene i polyfill necessari ad Angular a seconda del browser e della versione.

La cartella environments contiene i file `environment.ts` e `environment.prod.ts`, che specificano un oggetto JSON con i parametri dell'applicazione. Quando compiliamo in debug, viene usato il primo file mentre, se compiliamo per la produzione, viene usato il secondo file grazie a un parametro nel file `.angular-cli.json`.

La cartella assets contiene tutti i file necessari alla nostra applicazione per funzionare ed essere visualizzata correttamente come file JavaScript, CSS, immagini, font e così via. I file in questa cartella sono automaticamente inclusi nel pacchetto generato dalla compilazione grazie a un parametro impostato nel file `.angular-cli.json`.

Un'applicazione Angular è composta dai componenti elencati nella Tabella 21.1.

Tabella 21.1 – Componenti di un'applicazione Angular.

Componente	Descrizione
Component	Frammento di HTML e classe TypeScript che interagiscono per gestire una pagina dell'applicazione o una sua porzione
Service	Contiene il codice di business dell'applicazione. Tipicamente usata come interfaccia tra i component e le WebAPI
Pipe	Classe che permette di formattare l'output di una proprietà in binding (utile per formattare date, numeri e così via)
Directive	Classe che permette di aggiungere comportamenti a un component o un tag HTML
Module	Si tratta di un contenitore degli oggetti appena menzionati

Nella cartella app troviamo i vari componenti della nostra applicazione. La prima cosa che troviamo è il component `app.component` suddiviso in tre file con estensione .css, .html e .ts. Il file CSS specifica un file CSS che è valido solo all'interno del component (opzionale), il file HTML specifica il frammento di HTML che il component gestisce, mentre il file TS è la classe TypeScript con cui il frammento HTML comunica in binding.

`app.component` è il component lanciato dalla nostra applicazione in fase di startup e corrisponde quindi alla home. Angular prende il codice HTML del component e lo aggancia non al tag `body` della pagina index.html bensì all'interno del tag `<app-root>` visibile nel prossimo esempio.

Esempio 21.1

```
<body>
   <app-root>Loading...</app-root>
</body>
```

Ora che il frammento di HTML è stato attaccato alla pagina, possiamo metterlo in binding con il suo component. Nell'Esempio 21.2 vediamo sia il file HTML sia il file TS e notiamo come questi comunicano e quali funzionalità espongono.

Esempio 21.2 - TypeScript

```
import { Component } from '@angular/core';
@Component({

   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'app';
}
```

Esempio 21.2 - HTML

```
<div class='container-fluid'>
   <div class='row'>
      <div class='col-sm-3'>
         <app-nav-menu></app-nav-menu>
      </div>
      <div class='col-sm-9 body-content'>
         <router-outlet></router-outlet>
      </div>
   </div>
</div>
```

Questo esempio contiene molte funzionalità di Angular. Ma andiamo per passi, cominciando dal file TypeScript. Questo file contiene una semplice classe che ha una proprietà di nome `app` e che viene agganciata ad Angular grazie al decorator `@Component` che fa parte del framework e che viene importato nel file grazie all'utilizzo di `import`.

Il decorator espone le proprietà `selector`, che specifica il tag HTML associato al component, `templateUrl`, che specifica il nome del file HTML legato alla classe (template), e `styleUrls`, che specifica eventuali file CSS da associare al template. È grazie alla proprietà `selector` che, in fase di startup, Angular è in grado di collegare il tag `<app-root>` nel file `index.html` al component e utilizzarlo come home dell'applicazione.

Il template contiene esclusivamente codice HTML oltre a due tag sconosciuti: `<app-nav-menu>` e `<router-outlet>`. Il primo tag renderizza al suo interno un altro component che ha specificato il valore `app-nav-menu` nella proprietà `selector` del suo decorator. Questo component si trova nella cartella nav-menu e contiene il menu di navigazione. Il secondo è un tag Angular che specifica l'area in cui inserire il codice HTML dei component che carichiamo quando navighiamo nell'applicazione. Da questo deriva che `app.component` contiene il layout generale della nostra applicazione, in cui è compreso anche il menu.

Visto che app.component non è una vera e propria pagina dell'applicazione ma un contenitore, dobbiamo avere a disposizione un altro component verso cui eseguire la navigazione quando si carica l'app. Questo mapping lo specifichiamo nel modulo app.module, all'interno del quale specifichiamo i component della nostra applicazione e il routing, come viene mostrato nel seguente codice.

Esempio 21.3

```
@NgModule({
   declarations: [
      AppComponent,
      NavMenuComponent,
      HomeComponent,
      CounterComponent,
      FetchDataComponent
   ],
   imports: [
      BrowserModule.withServerTransition({ appId: 'ng-cli-universal' }),
      HttpClientModule,
      FormsModule,
      RouterModule.forRoot([
         { path: '', component: HomeComponent, pathMatch: 'full' },
         { path: 'counter', component: CounterComponent },
         { path: 'fetch-data', component: FetchDataComponent },
      ])
   ],
   providers: [],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

Un module è una classe che viene collegata ad Angular grazie al decorator @NgModule. Questo decorator espone la proprietà declarations, tramite la quale specifichiamo i component del module, la proprietà imports, tramite la quale specifichiamo quali altri moduli importiamo nel nostro per poterne utilizzare le classi, providers che contiene i service e bootstrap che specifica il component di startup (da utilizzare solo nel modulo principale).

Poiché la nostra applicazione naviga tra pagine, importiamo il module RouterModule e, tramite il suo metodo forRoot, creiamo un mapping tra url e component da caricare. Nel nostro caso, quando chiediamo la root del sito, viene caricato app-component e poi si naviga verso il component HomeComponent. Quando l'utente naviga verso l'url counter, viene caricato CounterComponent e, se si naviga verso l'url fetch-data, viene caricato FetchDataComponent.

Quest'ultimo component è interessante, in quanto mostra perfettamente come mettere in binding il template con i dati nel component presi da una WebAPI. Cominciamo dal codice del component.

Esempio 21.4

```
@Component({
   selector: 'app-fetch-data',
   templateUrl: './fetch-data.component.html'
})
export class FetchDataComponent {
   public forecasts: WeatherForecast[];
   constructor(http: HttpClient, @Inject('BASE_URL') baseUrl: string) {
      http.get<WeatherForecast[]>(baseUrl + 'api/SampleData/WeatherForecasts')
         .subscribe(result => this.forecasts = result);
   }
}
interface WeatherForecast {
   dateFormatted: string;
   temperatureC: number;
   temperatureF: number;
   summary: string;
}
```

Il component ha una proprietà forecasts che è un array di oggetti di tipi di tipo WeatherForecast. Nel costruttore della classe, accettiamo in input due parametri: uno di tipo HttpClient che rappresenta l'oggetto da usare per effettuare chiamate AJAX verso le WebAPI e l'altro di tipo string, che rappresenta l'url di base delle WebAPI. Questi parametri vengono iniettati nel costruttore del component dal motore di dependency injection di Angular.

Per recuperare i dati, usiamo il metodo get specificando tramite parametro generico il tipo di oggetto restituito dal server e passando in input l'url del servizio. Questo metodo restituisce un oggetto di tipo Observable (classe facente parte della libreria RxJS da

cui Angular dipende) e noi usiamo il metodo `subscribe` per ricevere le notifiche della risposta dal server. Quando il risultato arriva, viene invocato il callback passato in input al metodo `subscribe`, all'interno del quale valorizziamo la proprietà `forecasts` con il risultato della chiamata.

La classe HttpClient espone anche i metodi post, put, delete e altri ancora per effettuare ogni tipo di chiamata HTTP.

Una volta recuperati i dati dal servizio, dobbiamo visualizzarli sulla UI e questo è compito del template tramite il codice dell'Esempio 21.5.

Esempio 21.5

```
<p *ngIf="!forecasts"><em>Loading...</em></p>

<table class='table' *ngIf="forecasts">
   <thead>
      <tr>
         <th>Date</th>
      </tr>
   </thead>
   <tbody>
      <tr *ngFor="let forecast of forecasts">
         <td>{{ forecast.dateFormatted }}</td>
      </tr>
   </tbody>
</table>
```

La prima caratteristica di questo esempio sono le direttive `ngIf` e `ngFor` di Angular. La prima renderizza o no un tag se la condizione della direttiva è vera o falsa, mentre la seconda ripete il tag tante volte quando gli elementi contenuti nell'array che si sta ciclando nell'espressione passata in input alla direttiva. In questo esempio, la proprietà `forecasts` è `undefined` finché il servizio non risponde, quindi viene mostrato il messaggio di caricamento finché il servizio non risponde e poi, una volta che la proprietà `forecasts` è valorizzata con la risposta del servizio, il messaggio viene nascosto e viene mostrata la tabella. Tramite `ngFor` per ogni oggetto dell'array replichiamo il template, sfruttando la sintassi che prevede le parentesi graffe doppie per specificare il binding, come abbiamo già visto nel Capitolo 19 con Vue.js. Il risultato è visibile nella Figura 21.4.

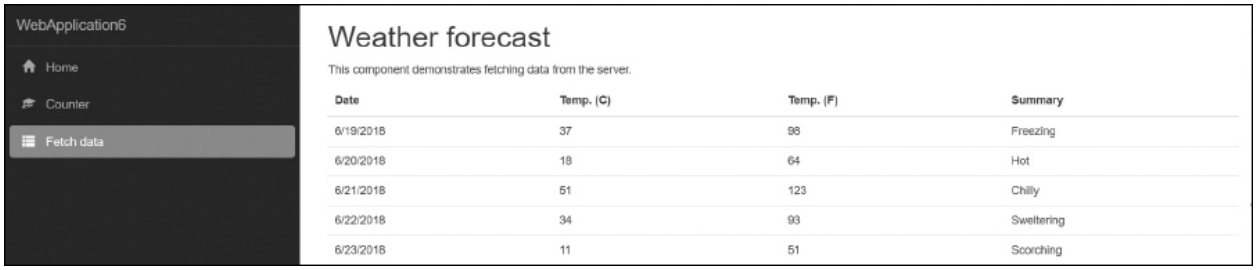


Figura 21.4 – L'applicazione mostra i dati ricevuti dal servizio.

Questo esempio mostra chiaramente come la separazione dei compiti tra component e template sia netta e come questi si parlino via binding. Tuttavia, abbiamo visto solo un rovescio della medaglia cioè il binding dal component verso il template, ma possiamo fare anche l'inverso, ovvero far reagire il component a un evento sulla UI come il click di un pulsante. Questo esempio è mostrato nel `CounterComponent` e ne mostriamo un estratto nel prossimo codice.

Esempio 21.6 - TypeScript

```
export class CounterComponent {
   public currentCount = 0;
   public incrementCounter() {
      this.currentCount++;
   }
}
```

Esempio 21.6 - HTML

```
<p>Current count: <strong>{{ currentCount }}</strong></p>
<button (click)="incrementCounter()">Increment</button>
```

Grazie all'utilizzo delle parentesi tonde, intorno alla direttiva `click` specifichiamo ad Angular che stiamo mettendo in binding un evento della UI con un metodo sul component. Successivamente, il metodo del component aggiorna il valore della proprietà `currentCount`, che essendo in binding viene aggiornata sulla UI.

Se vogliamo vedere in azione l'applicazione, tutto quello che dobbiamo fare è lanciarla con o senza debug e navigare. Quando lanciamo l'applicazione, il compilatore di Angular lancia prima il compilatore TypeScript per convertire il codice in JavaScript, poi compila il codice HTML e CSS per includerlo nei file JavaScript e, infine, copia questi file, la cartella assets e il file `index.html` in una cartella dist a cui il web server punta. Se vogliamo generare una build da mettere in produzione, ci basta entrare nella riga di comando e digitare il comando `ng build --prod`, che esegue gli stessi passi con la sola differenza di ottimizzare al massimo il codice per renderlo il più piccolo possibile.

In questa sezione abbiamo visto come creare un progetto con Angular e ASP.NET Core, partendo da zero, abbiamo visto come il progetto è strutturato e come funzionano binding e navigazione in Angular. Ovviamente in Angular c'è molto di più, come le form, la creazione di controlli custom e librerie, la creazione Progressive Web Application e altro ancora. Per ovvi motivi di spazio non ci è possibile approfondire Angular, per cui rimandiamo alla documentazione, che è disponibile all'indirizzo: `http://aspit.co/bpo`.

ReactJS

ReactJS è la libreria targata Facebook per creare SPA. Così come Angular, questa ibreria è largamente usata dagli sviluppatori web e per questo motivo Microsoft ha deciso di includerla all'interno di Visual Studio, sempre utilizzando il wizard di creazione di un progetto web e selezionando nella finestra della Figura 21.2 l'opzione React.js al posto di Angular. Il template dietro questa opzione genera un'applicazione con le stesse funzionalità di quella generata per Angular, quindi il risultato alla fine del processo di creazione è un progetto che ha lo stesso controller per le WebAPI e la cartella ClientApp che è come quella visibile nella Figura 21.5.

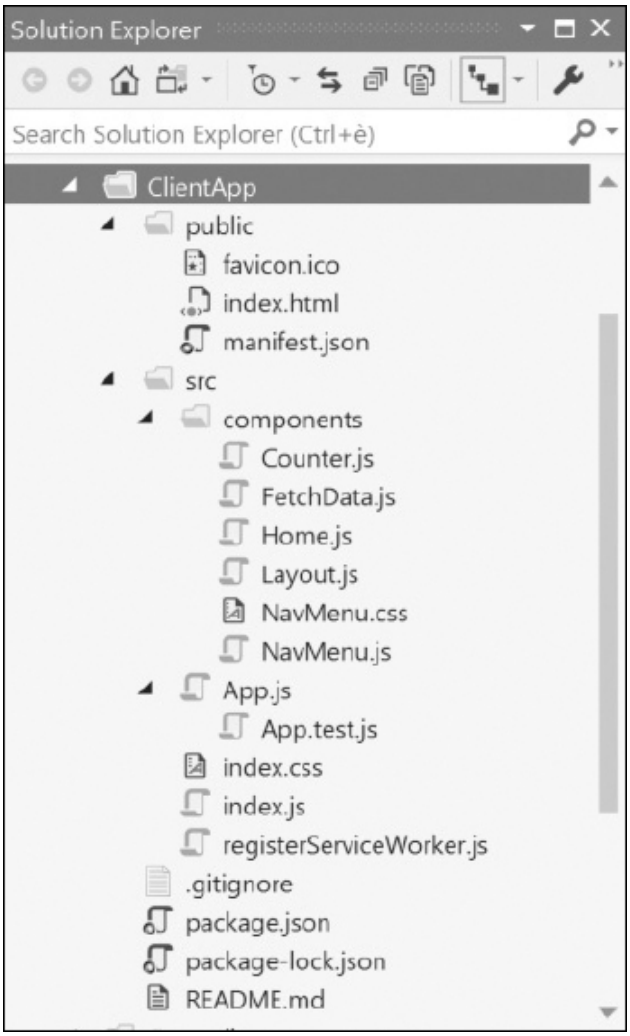


Figura 21.5 – La finestra Solution Explorer mostra l'applicazione ReactJS appena creata.

Nella root di ClientApp troviamo il file `package.json` che contiene i riferimenti agli strumenti e alle librerie da scaricare da NPM mentre nella cartella public troviamo alcuni file tra cui `index.html`, che rappresenta il master e che contiene nel tag `<body>` solo un tag `<div>` con id `root`, all'interno del quale girerà l'applicazione.

La cartella src contiene il vero codice dell'applicazione, a partire dal file `index.js`, che contiene il codice di startup mostrato nell'Esempio 21.7.

Esempio 21.7

```
const baseUrl =
   document.getElementsByTagName('base')[0].getAttribute('href');
const rootElement = document.getElementById('root');

ReactDOM.render(
<BrowserRouter basename={baseUrl}>
   <App />
</BrowserRouter>,
rootElement);
```

Il metodo `render` della classe `ReactDOM` renderizza il codice HTML al suo interno, sfruttando la sintassi JSX. Questa è una grossa differenza rispetto ad Angular, dove il codice HTML è in un template a parte. `<BrowserRouter>` è un tag che rappresenta un componente di ReactJS, all'interno del quale renderizzare le pagine dell'applicazione durante la navigazione, mentre `<app>` è un tag che rappresenta un componente custom dell'applicazione contenuto nel file `app.js`. Il secondo parametro è l'oggetto HTML all'interno del quale renderizzare il codice HTML passato come primo parametro.

Il file `app.js` rappresenta un componente in ReactJS e si tratta di una classe che estende la classe base `Component` di cui esegue l'override del metodo `render`, che restituisce il codice HTML del componente. Il metodo `render` del componente `App` è quello dell'Esempio 21.8.

Esempio 21.8

```
export default class App extends Component {
   displayName = App.name
   render() {
      return (
         <Layout>
            <Route exact path='/' component={Home} />
            <Route path='/counter' component={Counter} />
            <Route path='/fetchdata' component={FetchData} />
         </Layout>
      );
   }
}
```

Il tag `<Layout>` rappresenta un altro componente dell'applicazione che al suo interno prende la lista delle rotte dell'applicazione e il componente relativo a ogni rotta. `Layout` comprende il layout HTML dell'applicazione come l'intestazione, il menu e il punto in cui la pagina specificata dal routing deve essere renderizzata.

Anche in questo caso, le pagine sono rappresentate da due componenti `FetchData` e `Counter`. Il componente `FetchData` esegue la chiamata al server e mostra i dati a video. Quindi, è quello che più ci interessa per quanto riguarda l'interazione con ASP.NET Core. Vediamo il suo codice nell'Esempio 21.9.

Esempio 21.9

```
export class FetchData extends Component {
   constructor(props) {
      super(props);
      this.state = { forecasts: [], loading: true };
      fetch('api/SampleData/WeatherForecasts')
         .then(response => response.json())
         .then(data => {
            this.setState({ forecasts: data, loading: false });
         });
   }
   static renderForecastsTable(forecasts) {
      return (
         <table className='table'>
            <thead>
               <tr>
                  <th>Date</th>
               </tr>
            </thead>
         <tbody>
            {forecasts.map(forecast =>
               <tr key={forecast.dateFormatted}>
                  <td>{forecast.dateFormatted}</td>
               </tr>
```

```
            )}
         </tbody>
      </table>
     );
   }
   render() {
      let contents = this.state.loading
         ? <p><em>Loading...</em></p>
         : FetchData.renderForecastsTable(this.state.forecasts);
      return (
         <div>
            <h1>Weather forecast</h1>
            <p>This component demonstrates fetching data from the server.</p>
            {contents}
         </div>
      );
   }
}
```

Ogni componente in ReactJS ha uno stato accessibile tramite la proprietà `state`, manipolabile tramite il metodo `setState` e messo in binding con il codice HTML. In questo caso, nel costruttore impostiamo lo stato con un oggetto che ha la proprietà `loading`, che specifica se i dati sono stati caricati o no, e la proprietà `forecasts`, che contiene i dati recuperati dal server. Successivamente, invochiamo il metodo `fetch` passando in input l'url della WebAPI. Quando il server risponde, impostiamo la proprietà `forecasts` con i dati restituiti dal server e la proprietà `loading` a `true` per indicare che il caricamento è terminato.

Il metodo `render` renderizza un messaggio di attesa finché la proprietà `loading` è `false` e la tabella con i dati, una volta caricati. Il codice HTML è interpolato con codice JavaScript attraverso l'utilizzo delle parentesi graffe singole. Questo significa che, a differenza di Angular, non abbiamo direttive per simulare istruzioni `if`, `for` o altro ancora, ma utilizziamo direttamente il codice JavaScript. Per alcuni sviluppatori la sintassi JSX per descrivere il codice HTML è più chiara, mentre per altri lo è meno. Questo pesa molto sulla scelta tra un framework e l'altro.

Anche in questo caso, abbiamo visto solamente il binding dal componente verso la UI, ma ReactJS permette ovviamente il contrario cioè di invocare un metodo sul componente a seguito di un evento sulla UI. Anche in questo caso, ci viene in aiuto la sintassi che usa le parentesi graffe singole.

Esempio 21.10

```
export class Counter extends Component {
   constructor(props) {
      super(props);
      this.state = { currentCount: 0 };
      this.incrementCounter = this.incrementCounter.bind(this);
   }
   incrementCounter() {
      this.setState({
         currentCount: this.state.currentCount + 1
      });
   }
   render() {
      return (
         <div>
            <p>Current count: <strong>{this.state.currentCount}</strong></p>
            <button onClick={this.incrementCounter}>Increment</button>
         </div>
      );
   }
}
```

Anche in questo caso, la trattazione di ReactJS è limitata allo startup, al disegno della UI, al binding e all'interfacciamento con il server per lavorare con i dati. Ovviamente c'è un universo di funzionalità che per ovvi motivi non possiamo coprire ma che può essere approfondito direttamente sul sito di ReactJS, disponibile all'url: http://aspit.co/bpq.

Conclusioni

Le SPA sono un paradigma di sviluppo che sta prendendo sempre più piede nel mondo web. Ormai, quando si inizia un nuovo progetto, questo modello è una scelta da tenere sempre in considerazione in virtù del fatto che la sua maturazione è ormai a un livello tale da renderlo equiparabile in molti casi al modello ibrido.

La quantità e la qualità dei framework a disposizione è tale da non far rimpiangere la mancanza di un linguaggio fortemente tipizzato come C#. Inoltre, le specifiche di HTML, CSS e JavaScript sono in continua evoluzione il che continua ad aprire nuovi scenari in cui le SPA possono essere un player insostituibile, come nel caso delle Progressive Web Application.

Ovviamente non è tutto oro quel che luccica, quindi la scelta tra il modello SPA e il modello ibrido deve essere sempre attentamente valutata.

Con questo argomento abbiamo terminato la nostra trattazione relativa ad argomenti di sviluppo con ASP.NET Core, ma non è finita qui. Le nostre applicazioni, infatti, devono anche essere messe in produzione. Nel prossimo capitolo ci occuperemo proprio di questi argomenti.

22
Configurare e pubblicare un'applicazione ASP.NET Core

Nel corso del libro abbiamo affrontato tutti i temi principali relativi allo sviluppo di un'applicazione web, partendo dai concetti di MVC, arrivando alla definizione delle view, fino a parlare dei dati e di come localizzarli, esporli, proteggerli e consumarli. Tuttavia, nonostante tutti questi concetti siano risultati utili alla costruzione di una vera e propria applicazione, non possono essere messi in pratica e testati effettivamente fino a quando non si arriva alla fase del rilascio, in modo che chiunque possa vedere il prodotto realizzato.

Tipicamente, in passato, distribuire un'applicazione ASP.NET significava preparare un'ambiente con macchine Windows Server e Internet Information Server (IIS). Inoltre, era quasi dato per scontato che la versione del .NET Framework utilizzata dal team di sviluppo fosse la stessa disponibile nell'ambiente di produzione, soprattutto considerati i lunghi periodi tra il rilascio di una versione e la successiva. A oggi, però, sono cambiati tanti aspetti:

- Tempistiche di rilascio: i tempi che intercorrono tra la fase di sviluppo e il rilascio in produzione sono notevolmente ridotti per via di pratiche come continuous integration e continuous deployment.
- Ambienti multipli: è fondamentale rilasciare un prodotto che sia testato – possibilmente da più team – e funzionante (sviluppo, QA di diversi livelli) ma, soprattutto per via dei tempi ristretti, è bene testare le applicazioni in ambienti che non siano produzione (per evitare potenziali problematiche) ma che siano piuttosto simili per effettuare simulazioni di upgrade e downgrade.
- L'hardware e il software: .NET Core, al contrario di .NET Framework, è in grado di eseguire le applicazioni anche in ambienti Linux e macOS, quindi non è più possibile dare per scontata la presenza di Windows Server e di IIS; inoltre, il framework potrebbe non essere installato fisicamente sulla macchina ma distribuito con l'applicazione stessa.
- Tipologia di lavoro: il team di lavoro moderno e ideale si sta sempre di più spostando verso il mondo agile e DevOps, quindi diventa fondamentale avere conoscenze sia di sviluppo software sia di gestione e manutenzione (operations) dell'infrastruttura sulla quale viene eseguito il software, per questo sono emersi temi fondamentali come i container e il cloud.

All'interno di questo capitolo affronteremo proprio il concetto relativo al deployment e al delivery delle applicazioni e parleremo nel dettaglio di come le novità introdotte da .NET Core (e ASP.NET Core) aiutino a risolvere le problematiche sollevate dagli aspetti evidenziati. Prima di procedere al rilascio dell'applicazione, però, è lecito domandarsi in quale ambiente l'applicazione stessa verrà distribuita: di fatto, è possibile prevedere un cambio del comportamento sulla specifica dell'ambiente.

Modificare il comportamento secondo l'ambiente di destinazione

Come abbiamo già descritto nei primi capitoli di questo libro, ci sono casi in cui il comportamento dell'applicazione può essere diverso a seconda dell'ambiente in cui viene eseguito: l'ambiente di sviluppo (o `Development`) potrebbe prevedere un logging molto avanzato e continuo su ogni singola chiamata, mentre l'ambiente di produzione potrebbe avere solo un logging sommario per non rallentare l'esecuzione, così come nell'ambiente di controllo qualità si potrebbe aver bisogno di un logging di tipo misto, in cui ha più senso verificare il flusso delle operazioni effettuate dagli utenti rispetto al sapere il dettaglio di risposta di ogni singola riga di codice. In produzione si vorrà avere la minification o, ancora, ci saranno endpoint differenti in base all'ambiente per richiamare una WebAPI oppure un background service. ASP.NET Core mette a disposizione gli environment, che permettono proprio di distinguere, a livello di codice, l'ambiente di esecuzione a seconda di quanto viene definito dagli sviluppatori oppure da operations direttamente sulla macchina fisica.

Come abbiamo già avuto modo di vedere in precedenza, gli environment di default sono `Development`, `Staging` e `Production` che vengono esposti direttamente dal framework tramite l'interfaccia `IHostingEnvironment` e, in particolare, tramite la proprietà `EnvironmentName`, ma, poiché ci sono scenari che richiedono configurazioni e ambienti molto più complessi, è anche possibile andare a creare ambienti personalizzati.

Esempio 22.1

```
public class MyClass
{
   public MyClass(IHostingEnvironment en)
   {
      // controlliamo l'ambiente
      if (en.IsEnvironment("MyCustomEnvironment")
      {
         // l'ambiente è identificato, eseguiamo operazioni specifiche...
      }
   }
}
```

Nell'Esempio 22.1 si può notare come sia possibile identificare un ambiente personalizzato per poter eseguire le operazioni specifiche relative a quell'environment. Non tutta la configurazione, però, è strettamente legata ai servizi che devono essere erogati (come, per esempio, il cambio di URL tra ambiente di sviluppo e di produzione), ma può anche essere relativa all'infrastruttura su cui viene eseguita, come per esempio l'uso di Kestrel piuttosto che IIS per il web server, l'entrypoint dell'applicazione stessa o, ancora, l'uso di variabili d'ambiente personalizzate. Tutte queste opzioni possono essere specificate all'interno del file `launchSettings.json` nella cartella `Properties` del progetto, come mostrato nell'Esempio 22.2.

Esempio 22.2

```
{
   "iisSettings": {
      "windowsAuthentication": false,
         "anonymousAuthentication": true,
         "iisExpress": {
            "applicationUrl": "http://localhost:43421",
            "sslPort": 44300
         }
      },
      "profiles": {
         "IIS Express": {
            "commandName": "IISExpress",
            "launchBrowser": true,
            "environmentVariables": {
               "ASPNETCORE_ENVIRONMENT": "Development"
            }
         },
         "Capitolo22": {
            "commandName": "Project",
            "launchBrowser": true,
            "applicationUrl": "https://localhost:5001;http://localhost:5000",
            "environmentVariables": {
               "ASPNETCORE_ENVIRONMENT": "Development",
               "ASPNETCORE_CUSTOM_VALUE": "Custom",
         }
      }
   }
}
```

Tra le variabili d'ambiente esposte dall'Esempio 22.2 si può notare la sola `ASPNETCORE_ENVIRONMENT` che rappresenta proprio l'`EnvironmentName` di `IHostingEnvironment` per il profilo "IIS Express", mentre per il profilo "Capitolo22" è stata aggiunta anche una variabile custom. Tra le altre proprietà esposte da questo file di configurazione è bene evidenziare `commandName` poiché, durante l'esecuzione del comando `dotnet run`, verrà fatto il discovery di tutti i profili e quindi verrà scelto il primo con il valore impostato a `Project`. I profili elencati all'interno di questo file sono anche utilizzati da Visual Studio per far partire l'applicazione web nel modo corretto, come viene illustrato nella Figura 22.1.

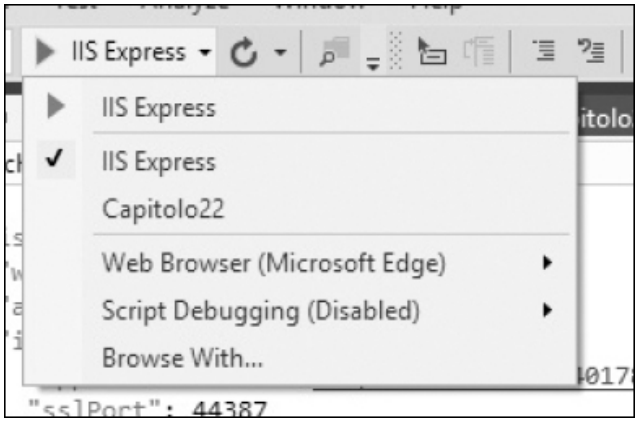


Figura 22.1 – I profili creati all'interno del file `launchSettings.json` vengono proposti da Visual Studio per l'esecuzione dell'applicazione web.

Il comando `dotnet run`, nello specifico, esegue questa serie di operazioni in sequenza, che vengono mostrate nella Figura 22.2: recupera il file `launchSettings.json` per capire quale profilo avviare, quindi legge i valori delle variabili d'ambiente e li sostituisce a

quelli eventualmente definiti tramite variabili d'ambiente a livello di sistema, poi recupera il valore dell'environment, esegue il `Main` della classe `Program` che avvia il web server e apre le porte necessarie e, infine, carica la configurazione opportuna.

```
Windows PowerShell
PS C:\book\Capitolo22> dotnet run
Uso delle impostazioni di avvio di C:\book\Capitolo22\Properties\launchSettings.json...
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\Matteo\AppData\Local\ASP.NET\DataProtection-Keys'
as key repository and Windows DPAPI to encrypt keys at rest.
Hosting environment: Development
Content root path: C:\book\Capitolo22
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figura 22.2 – L'esecuzione del comando `dotnet run` forza la lettura delle impostazioni di avvio.

La variabile d'ambiente `ASPNETCORE_ENVIRONMENT` può essere impostata a diversi livelli, in modo dipendente dal sistema operativo e dalla durata prevista. Per avere un'impostazione temporanea a livello di finestra/esecuzione corrente, su Windows, per esempio, può essere impostata tramite i comandi `setx` e `$Env` rispettivamente per Command Line e PowerShell, mentre per macOS e distribuzioni Linux può essere impostata tramite il comando `export`. Purtroppo, però, l'esecuzione di questi comandi è limitata, quindi in ambienti di produzione o in scenari in cui l'applicazione può andare in crash non è l'ideale, dato che riavviandosi perderebbe l'ambiente precedentemente impostato. Per questo è necessario impostare la variabile direttamente a livello di sistema operativo, tramite il `bash_profile` di Linux e macOS, oppure tramite le variabili d'ambiente di Windows, come illustrato nella Figura 22.3.

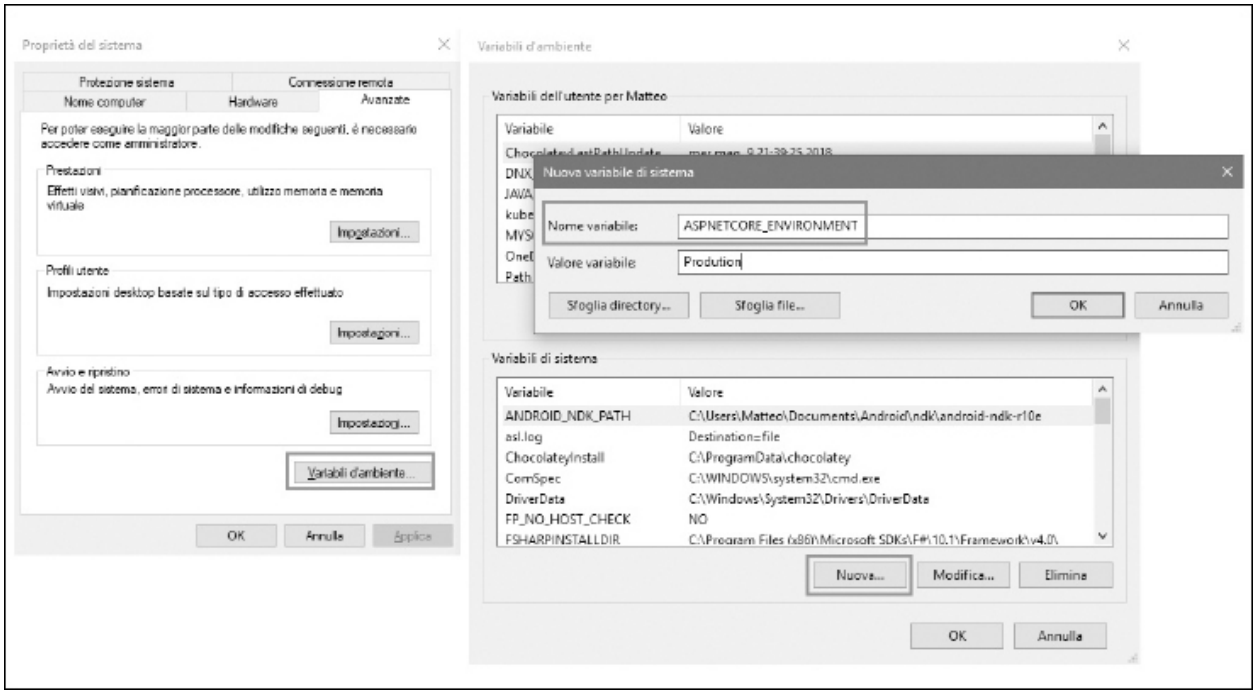


Figura 22.3 – L'impostazione di una variabile di sistema su sistema operativo Windows avviene tramite il menù Proprietà di sistema -> Variabili d'ambiente -> Nuova variabile di sistema.

Qualora l'applicazione venga eseguita in ambiente Windows con IIS, allora è anche possibile specificare l'environment e le altre variabili d'ambiente tramite il `web.config`, come dimostra l'Esempio 22.3.

Esempio 22.3

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.webServer>
      <handlers>
         <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
               resourceType="Unspecified" />
```

```
        </handlers>
        <aspNetCore processPath="dotnet"
                        arguments=".\Capitolo22.dll"
                        stdoutLogEnabled="false"
                        stdoutLogFile=".\logs\stdout">
        <environmentVariables>
           <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
           <environmentVariable name="ASPNETCORE_CUSTOM_VALUE" value="Custom" />
        </environmentVariables>
      </aspNetCore>
   </system.webServer>
</configuration>
```

All'interno di un progetto vuoto ASP.NET Core MVC, la prima volta in cui si incontra l'uso della variabile `ASPNETCORE_ENVIRONMENT` è nel metodo `Configure` della classe `Startup`, in cui l'uso della pagina di errore dettagliato e di HSTS vengono aggiunti solamente per gli ambienti di sviluppo e produzione rispettivamente, come è illustrato nell'Esempio 22.4.

Esempio 22.4

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
   }
   else
   {
      app.UseExceptionHandler("/Error");
      app.UseHsts();
   }
   app.UseMvc();
}
```

Se però l'applicazione cresce e diventa via via più complessa, cresce il numero di environment oppure il numero di servizi e l'esecuzione degli stessi dipende dagli ambienti, può diventare molto complicato riuscire a gestire tutto il flusso di inizializzazione previsto dalla classe `Startup` solo con una serie di statement condizionali. Proprio per questo motivo, i metodi `Configure` e `ConfigureServices` in realtà sono solo un'astrazione dei veri nomi, rappresentati da `Configure{environment-name}` e `Configure{environment-name}Services`.

Esempio 22.5

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddMvc();
}

public void ConfigureDevelopmentServices(IServiceCollection services)
{
   services.AddAuthentication().AddMvc();
}
```

Come si può notare nell'Esempio 22.5, infatti, anziché usare un `if` per controllare se l'ambiente di esecuzione è `Development`, per aggiungere l'autenticazione si è preferito utilizzare la separazione tra i due metodi. Poiché la stessa cosa può avvenire per entrambi i metodi `Configure` e `ConfigureServices` per ogni environment configurato, si corre il rischio di aggiungere troppa complessità e illeggibilità alla classe di `Startup` ma, proprio per questo motivo, questa naming convention è disponibile anche per la classe di inizializzazione, come è illustrato nell'esempio seguente.

Esempio 22.6

```
public class StartupDevelopment
{
   public StartupDevelopment(IConfiguration configuration)
   {
      // invocato quando l'ambiente è Development...
   }
}

public class Startup
```

```
{
   public Startup(IConfiguration configuration)
   {
      // invocato quando l'ambiente non è Development...
   }
}
```

Se si decide di creare classi `Startup` per ogni environment, come viene mostrato nell'Esempio 22.6, però, è bene andare a modificare il metodo `CreateWebHostBuilder` della classe `Program` perché questo fa un uso tipizzato della classe `Startup` che va cambiato per renderlo generico e specifico dell'ambiente di esecuzione, come dimostrato nell'Esempio 22.7.

Esempio 22.7

```
public class Program
{
   public static void Main(string[] args)
   {
      CreateWebHostBuilder(args).Build().Run();
   }
   public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
               .UseStartup(Assembly.GetExecutingAssembly().FullName);
            //.UseStartup<Startup>();
}
```

Nell'Esempio 22.7 è messo in evidenza che non è più possibile indicare in modo esplicito il nome della classe `Startup` perché deve essere calcolato secondo il valore della variabile d'ambiente. Pertanto è necessario fare uso di un override del metodo `UseStartup`, che permette di specificare l'assembly in cui si troverà, quindi, il motore di bootstrapping ASP.NET Core farà il resto. Volendo, è possibile testare il funzionamento andando a impostare il parametro di environment direttamente nella creazione dell'host, come mostrato nell'Esempio 22.8.

Esempio 22.8

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
   WebHost.CreateDefaultBuilder(args)
            .UseEnvironment("Development")
           .UseStartup(Assembly.GetExecutingAssembly().FullName);
```

Come abbiamo già avuto modo di vedere all'interno del Capitolo 3, tutta la configurazione specifica per ogni ambiente viene prelevata dai file `appsettings.{environment-name}.json` e può essere letta all'interno dell'applicazione sia tramite `IConfiguration` sia in maniera fortemente tipizzata. Questo, unito a quanto visto in questo stesso capitolo per le variabili d'ambiente, però, non garantisce alcun tipo di protezione dei dati: infatti, tutte le chiavi, come per esempio connection string per il database ed eventuali password, sono esposte in chiaro e sono leggibili da chiunque. Pertanto è necessario un sistema basato a secret che assicuri un minimo di sicurezza, come abbiamo già visto nei capitoli precedenti.

Una volta definiti gli ambienti di esecuzione, parametrizzato l'applicazione per poter lavorare con environment diversi e aggiunto il supporto alle secret e ad Azure Key Vault, per poter lavorare in sicurezza con dati che non devono essere esposti all'esterno o vulnerabili tramite la macchina stessa, è il momento di pensare al deployment, ovvero al rilascio in un ambiente dedicato dell'applicazione web creata.

Pubblicazione e hosting

Durante l'introduzione di questo capitolo abbiamo illustrato come e quanto sono cambiate le pratiche per il rilascio delle applicazioni per via di diversi fattori quali il tempo, le competenze tecniche e la tipologia di framework. Nonostante tutti questi cambiamenti, però, il flusso in linea generale non cambia e si mantiene su tre aspetti:

- Pubblicazione: il codice sorgente deve essere compilato e distribuito in una cartella del server.
- Scelta del process manager: la scelta è sempre stata implicita su Windows, con IIS per ASP.NET, ma con ASP.NET Core c'è bisogno di un process manager anche per gli ambienti macOS e Linux che sia in grado di riavviare l'applicazione in caso di malfunzionamenti e che riesca a gestire le richieste in ingresso, rimandandole all'applicazione stessa.
- Configurazione del reverse proxy: è una scelta opzionale, da fare in quei casi in cui non si vuole esporre il servizio in modo diretto; così facendo, il reverse proxy prenderà le richieste e le rimanderà al web server in un secondo momento.

La pubblicazione è già stata analizzata nel Capitolo 2 e, sostanzialmente, permette, tramite il comando `dotnet publish` della CLI, di generare DLL, file e dipendenze in una cartella specifica per la pubblicazione, in maniera tale che non venga copiato in alcun modo il codice sorgente originale. Abbiamo anche già visto le differenze tra un deployment di tipo self-contained, in cui nella cartella di pubblicazione

verrà aggiunto il runtime, creando così un pacchetto di pubblicazione più grande, rispetto alla pubblicazione classica di tipo framework-dependent, in cui si dà per scontato che il runtime sia già installato fisicamente sulla macchina server di destinazione, e pertanto non discuteremo di ulteriori dettagli.

La scelta del web server, invece, è importante, non solo perché non né si può fare a meno, dato che il sistema in-process serve a rimandare le richieste HTTP verso `HttpContext` all'interno dell'applicazione, ma anche perché prima di decidere quale web server utilizzare, è necessario capire il sistema operativo di destinazione, ed è una scelta strategica. Poiché ASP.NET Core è in grado di funzionare cross-platform, Microsoft ha deciso di rilasciare due implementazioni:

- Kestrel: di default, funziona anche cross-platform.
- HTTP.sys: chiamato in passato WebListener, è l'implementazione esclusiva per Windows, basata sul kernel driver HTTP.sys e HTTP server API.

Abbiamo già descritto i vantaggi con una tabella comparativa a inizio libro nell'uso di uno, dell'altro, o di un server personalizzato costruito sulla base dell'interfaccia `IServer` ma, in generale, poiché Kestrel è di default e il funzionamento è cross-platform, sarà quello che utilizzeremo in futuro.

L'uso di un reverse proxy, inoltre, non è strettamente necessario, poiché Kestrel è considerato, dalla versione 2 di ASP.NET Core production-ready ma ci sono decine di casi in cui ha comunque senso utilizzarlo: limitare l'esposizione dell'host, aggiungere un nuovo strato di sicurezza, limitare il numero di chiamate e il traffico generato e semplificare il load balancing sono solo alcuni esempi. Proprio per queste motivazioni, vedremo ora come configurare NGINX e Apache come reverse proxy per ambienti Linux e macOS.

Configurazione di NGINX

NGINX è un web server molto leggero, che garantisce ottime prestazioni per via del suo approccio asincrono basato su eventi generati dalle richieste HTTP che arrivano in ingresso ed è ottimo per funzionare come reverse proxy con Kestrel. Inoltre, funziona in ambienti Unix/Linux, BSD (macOS) e Windows, ed è perciò in grado di fornire quella componente cross-platform che abbiamo ricercato continuamente all'interno del libro. Per vedere un aspetto diverso e per mantenere una certa semplicità, le demo successive saranno costruite secondo un ambiente macOS con una sola istanza di NGINX non replicata, ma tutto il codice e le configurazioni che vedremo saranno portabili con pochi cambiamenti all'interno degli altri sistemi operativi e delle eventuali repliche.

Considerando che, una volta configurato il reverse proxy, le richieste passeranno prima da NGINX e poi verranno rimandate all'applicazione ASP.NET Core, è necessario configurare opportunamente gli header di forward, altrimenti, qualora nelle applicazioni web ci siano middleware o logiche che leggono questi valori, come per esempio sistemi di tracing delle request, questi potrebbero non funzionare più correttamente. Questo è particolarmente utile anche in scenari in cui si vuole avere un trace completo delle richieste per capire, potenzialmente, che cosa non stia funzionando a livello di routing.

Esempio 22.9

```
public void Configure (IApplicationBuilder app)
{
   app.UseForwardedHeaders(new ForwardedHeadersOptions
   {
      ForwardedHeaders = ForwardedHeaders.XForwardedFor |
         ForwardedHeaders.XForwardedProto;
   });
   app.UseMvc();
}
```

L'Esempio 22.9 mette in evidenza come tramite il metodo `Configure` si possa agire sulla configurazione di `UseForwardedHeaders` per specificare di rimandare gli header `X-Forwarded-For` (che conterrà l'indirizzo IP di origine della richiesta) e `X-Forwarded-Proto` (che conterrà invece il protocollo HTTP o HTTPS).

Una volta che abbiamo preparato l'applicazione a ricevere le richieste, arriva il momento di installare e configurare il web server: per l'installazione si può fare uso del package manager HomeBrew su macOS, con il comando `brew install nginx`, piuttosto che di APT per Linux con il comando `apt-get install nginx`. Per testare il funzionamento, è necessario avviare il web server tramite il comando `sudo nginx start` e, se questo è stato installato correttamente, partirà con la sua configurazione di default, mostrando una pagina HTML statica al raggiungimento dell'indirizzo `http://localhost:8080`. Una volta verificato il corretto funzionamento del servizio out-of-the-box, è necessario istruirlo per lavorare come proxy e quindi rimandare le chiamate. Per farlo, sarà sufficiente modificare la configurazione di default creata da NGINX nel file/etc/nginx/nginx.conf, come mostrato nell'Esempio 22.10.

Esempio 22.10

```
worker_processes 1;

events {
    worker_connections 1024;
}
```

```
http {
    include mime.types;
    default_type application/octet-stream;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;
     location / {
        proxy_pass http://localhost:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection keep-alive;
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
     }
   }
}
```

La maggior parte della configurazione esposta dall'Esempio 22.10 non è cambiata rispetto a quanto è previsto di default dal server. I cambiamenti si trovano solamente nella configurazione della porta 80 anziché 8080 e nella configurazione di tutti gli attributi proxy necessari a rimandare la chiamata all'applicazione. In particolare, poiché il server coincide con la macchina stessa, è stato aggiunto localhost, ma nulla vieta, una volta configurato SSL, di aggiungere il proprio dominio "mydomain. com". Le richieste verranno poi passate all'applicazione ASP.NET Core tramite `proxy_pass` che è stato impostato sempre su localhost nella porta 5000 (default di ASP.NET Core). Una volta salvata la configurazione appena impostata, è necessario rilanciare il web server con il comando `sudo nginx restart` e avviare l'applicazione web con il comando `dotnet nome.dll`: se il tutto è stato rappresentato correttamente, l'applicazione sarà raggiungibile sia tramite `http://localhost:5000`, ovvero tramite Kestrel, sia tramite `http://localhost`, `ovvero in reverse proxy da NGINX`.

Nonostante il sistema sia funzionante, a tutti gli effetti c'è ancora un'operazione che è stata eseguita manualmente, ovvero l'avvio dell'applicazione web; pertanto, non si stanno sfruttando i vantaggi di riavvio automatico visti in precedenza per i process manager. Per farlo, bisogna fare uso di launchd (o systemd su Linux) per creare e registrare servizi. Ogni servizio, in ambiente macOS, è rappresentato da un file con estensione `.plist`, in cui sono specificate tutte le proprietà relative a quello che vogliamo venga fatto in completa autonomia, come illustrato nell'Esempio 22.11.

Esempio 22.11

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-
1.0.dtd">
<plist version="1.0">
<dict>
      <key>KeepAlive</key>
      <true/>
      <key>Label</key>
      <string>Capitolo22</string>
      <key>ProgramArguments</key>
      <array>
              <string>/usr/local/share/dotnet/dotnet</string>
              <string>Capitolo22.dll</string>
      </array>
      <key>RunAtLoad</key>
      <true/>
      <key>StandardErrorPath</key>
      <string>/tmp/dotnet-api-sterr.log</string>
      <key>StandardOutPath</key>
      <string>/tmp/dotnet-api-stdout.log</string>
      <key>UserName</key>
      <string>root</string>
      <key>WorkingDirectory</key>
      <string>/Users/aspitalia/Desktop/Capitolo22/bin/Release/netcoreapp2.1/
publish</string>
      <key>EnvironmentVariables</key>
      <dict>
              <key>ASPNETCORE_ENVIRONMENT</key>
```

```
                <string>Production</string>
        </dict>
 </dict>
 </plist>
```

Il file mostrato nell'Esempio 22.11 viene salvato all'interno del percorso/Library/LaunchDaemons, così che, una volta registrato il servizio, al riavvio della macchina stessa venga avviato in automatico. All'interno di questo file vengono descritti: il nome del servizio con l'attributo `Label`, qual è il comando che deve essere avviato con i suoi parametri (ovvero `dotnet Capitolo22.dll`) con l'attributo `ProgramArguments`, i percorsi per i log di stdout e stderr, la working directory (che corrisponde al percorso in cui ci sarà la DLL che deve essere eseguita, le variabili d'ambiente e, infine, lo username che può avviare il servizio (`root` è l'unica opzione disponibile in LaunchDaemons). Per avviare il servizio, è necessario lanciare in sequenza i comandi mostrati nell'Esempio 22.12.

Esempio 22.12

```
cd /Library/LaunchDaemons
sudo launchctl load -w Capitolo22.plist
sudo launchctl start Capitolo22
```

Una volta registrato il servizio, si può controllare che sia effettivamente partito lanciando il comando `sudo launchctl list`, come mostrato nella Figura 22.4.

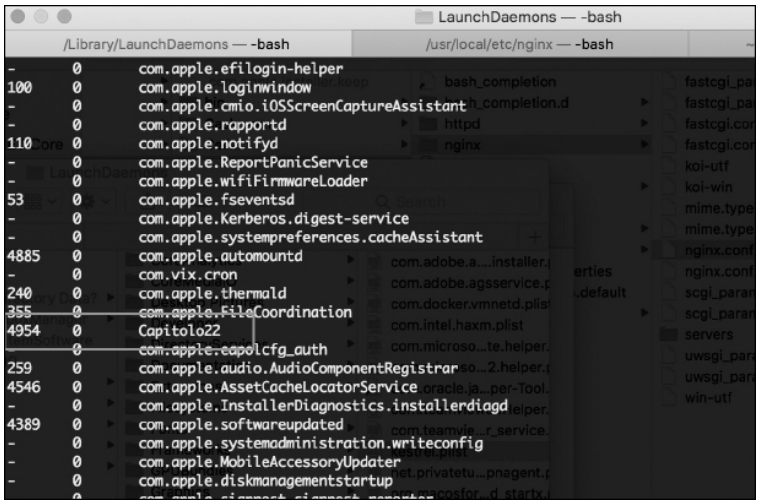


Figura 22.4 – Elenco dei servizi registrati come LaunchDaemons in macOS.

Qualora il servizio sia stato registrato correttamente, come mostrato nella Figura 22.4, con un PID associato, allora la configurazione di NGINX come reverse proxy sarà completata, totalmente automatizzata e resiliente in caso di errori o crash applicativi. Per testimoniare il corretto funzionamento, è sufficiente navigare all'indirizzo `http://localhost` e dovremmo vedere nuovamente il sito web ASP.NET Core, come illustrato nella Figura 22.5.

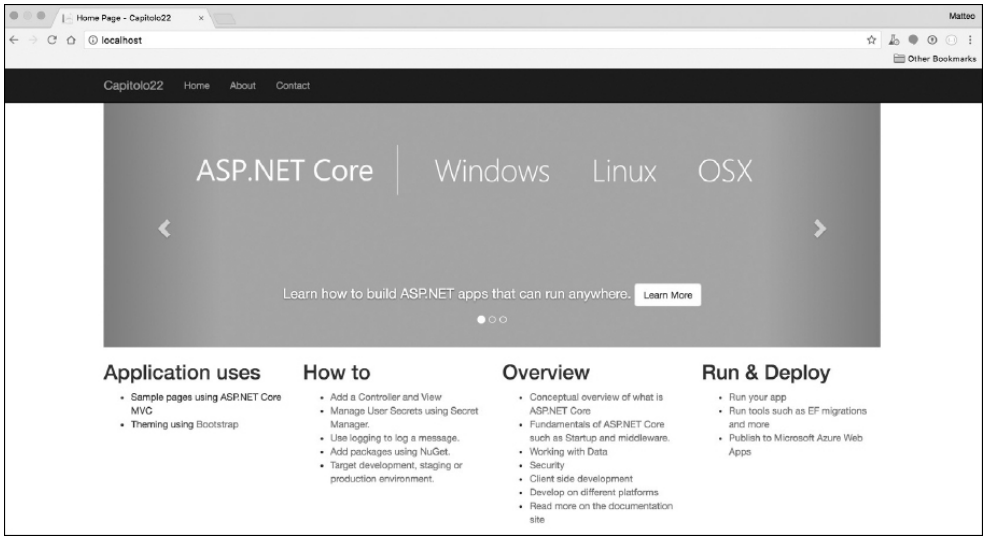


Figura 22.5 – Home page di un'applicazione web ASP.NET Core esposta in reverse proxy tramite NGINX e il servizio di restart automatico.

Poiché l'applicazione viene avviata in autonomia, non ci sarà a disposizione la console per mostrare i log ma, per via del fatto che sono stati esplicitati nel file del servizio `Capitolo22.plist`, sarà sufficiente lanciare il comando `tail -f /tmp/dotnet-api-stdout.log` per vedere lo stream dello standard output e `tail -f /tmp/dotnet-api-stderr.log` per vedere gli eventuali errori generati.

Nella configurazione di base documentata nell'Esempio 22.10, però, manca tutta la parte relativa alla sicurezza stessa del server, e pertanto è necessario andare ad aggiungere sul nodo alcuni header fondamentali, come illustrato nell'esempio seguente.

Esempio 22.13

```
server {
            listen *:443 ssl;
            add_header Strict-Transport-Security "max-age=63072000";
            add_header X-Frame-Options DENY;
            add_header X-Content-Type-Options nosniff;
}
```

Nell'Esempio 22.13 si può notare come siano stati aggiunti tre header:

- HTTP Strict-Transport-Security: HSTS è attivo di default in tutti i progetti a partire da ASP.NET Core 2.1 e permette di gestire tutte le richieste inviate dai client su HTTPS; per questo va accompagnato da certificati SSL e limiti di durata tramite `max-age`.
- X-Frame-Options: se impostato con valore `DENY` garantisce protezione dagli attacchi di clickjacking, in quanto non permette l'uso di frame con link a fonti esterne.
- X-Content-Type-Options: se impostato con valore `nosniff`, garantisce protezione dagli attacchi di MIME-Type sniffing, in cui il browser potrebbe cambiare l'header Content-Type e interpretare un oggetto di un tipo come di un altro.

La configurazione del server, poiché in questo caso risponderà sulla porta 443 anziché sulla porta 80, come abbiamo visto in precedenza, può essere affiancata alla configurazione mostrata nell'Esempio 22.10 e il sistema risponderà in reverse proxy sempre in modo adeguato, rimandando le richieste su HTTPS. Dal momento che il resto della configurazione del server dipende principalmente dalle proprie esigenze di produzione, non affronteremo temi più avanzati nel corso di questo capitolo ma rimandiamo alla documentazione ufficiale disponibile su `http://aspit.co/bot` e proseguiamo vedendo un web server alternativo, come Apache.

Configurazione di Apache HTTP Server

Apache HTTP Server (più comunemente Apache o httpd) rappresenta il secondo web server che si può sfruttare per applicare la configurazione di reverse proxy verso Kestrel, dato che anche questo è disponibile su più piattaforme, tra cui Linux, Windows e macOS. A oggi è tra i web server più utilizzati, probabilmente per via del fatto che il suo sviluppo è completamente open-source attraverso la Apache Software Foundation e che il primo rilascio fu effettuato nell'ormai lontano 1995. A livello puramente architetturale, differisce da NGINX per via del fatto che è completamente modulare: ogni componente è indipendente ed è in grado di gestire una singola funzionalità ma tutto il coordinamento tra i vari moduli avviene tramite il core, che prende in ingresso le richieste e le gira in sequenza a tutti i moduli che sono stati registrati. Poiché il core, per funzionare, ha bisogno continuamente di avere accesso a tutti i moduli, è necessario avere un servizio che tramite polling interroghi continuamente le singole componenti per aggiornare il loro stato, e per questo è meno performante rispetto a NGINX. A livello concettuale, quanto visto per NGINX non cambia rispetto a quanto vedremo con Apache: una volta effettuata l'installazione del web server, sarà necessario andare a cambiare la configurazione secondo le proprie esigenze e registrare lo stesso servizio già visto nell'Esempio 22.11 per ottenere lo stesso risultato. Anche quanto visto sulla configurazione e sulla pubblicazione dell'applicazione web stessa non cambia, e pertanto rimane consigliato abilitare il forwarding degli header, come è illustrato nell'Esempio 22.9.

L'installazione del server Apache dipende dal sistema operativo che si ha a disposizione ma, per coerenza, continueremo a utilizzare macOS per gli esempi a seguire. Pertanto si può fare uso del package manager HomeBrew e lanciare il comando `brew install httpd` (o `yum install httpd mod_ssl`) per procedere all'installazione nel caso in cui non sia già disponibile nella macchina server. Per verificare che il tutto sia stato configurato correttamente, si può lanciare il comando `sudo apachectl start` e aprire il browser in `http://localhost:8080` per vedere la prima pagina, come è illustrato nella Figura 22.6.

Verificato che il server è in grado di partire, esattamente com'è stato fatto in precedenza per NGINX, è necessario configurare il web server per il reverse proxy. In questo caso, poiché il sistema è composto da moduli, bisogna andare a modificare un modulo chiamato `httpd-vhosts.conf` contenuto nella cartella extra a partire dal percorso in cui è stato installato il web server (per esempio: /etc/local/ httpd) come è evidenziato nell'Esempio 22.14.

Figura 22.6 – La home page di Apache è una pagina `index.html` disponibile in un percorso variabile in base all'installazione effettuata (su macOS di default è /etc/local/var/www/ ma può essere cambiata nella configurazione di httpd) e pertanto può essere personalizzata.

Esempio 22.14

```
<VirtualHost *:8080>
      ProxyPreserveHost On
      ProxyPass / http://127.0.0.1:5000/
      ProxyPassReverse / http://127.0.0.1:5000/
      ServerName http://127.0.0.1
      ServerAlias localhost
      ErrorLog /tmp/error.log
      CustomLog /tmp/info.log common
</VirtualHost>
```

Come si può notare, nell'Esempio 22.14 sono stati registrati per il proxy gli indirizzi verso l'applicazione ASP.NET Core, che rimane in ascolto sulla porta 5000 (per HTTP), quindi, quando verrà fatta di nuovo la chiamata a `http://localhost:8080`, verremo reindirizzati all'applicazione e non più alla pagina di benvenuto vista nella Figura 22.7. Purtroppo, al momento, l'applicazione non è ancora visibile perché mancano due dettagli fondamentali: il primo è configurare il tutto in modo che il modulo httpd-vshosts venga caricato, il secondo invece è riavviare httpd con la nuova configurazione e il nuovo modulo. Per comunicare al core che il modulo `httpd-vhosts` deve essere caricato, bisogna andare a modificare la configurazione contenuta nel file `httpd.conf` decommentando le righe di codice mostrate nell'Esempio 22.15.

Esempio 22.15

```
LoadModule proxy_html module lib/httpd/modules/mod_proxy_html.so
LoadModule proxy_module lib/httpd/modules/mod_proxy.so
Include /usr/local/etc/httpd/extra/httpd-vhosts.conf
```

Le righe mostrate nell'Esempio 22.15 istruiscono il core su quale configurazione debba essere caricata ma, poiché all'interno dell'host vengono richiesti attributi relativi ai proxy per rimandare le chiamate, devono essere caricati tutti i moduli associati. A questo punto è sufficiente riavviare il web server con il comando `sudo httpd restart` per fare in modo che prenda la nuova configurazione per core e tutti i suoi componenti aggiuntivi per vedere a tutti gli effetti l'index dell'applicazione web ASP.NET Core.

Per quanto riguarda il resto della configurazione, valgono le stesse considerazioni già fatte per NGINX: se si vuole che il processo parta in automatico e non ci sia più il bisogno di eseguire a mano il comando `dotnet run`, allora si può registrare lo stesso servizio già discusso nell'Esempio 22.11 mentre, se si preferisce avere l'applicazione con HTTPS, sono comunque obbligatori i certificati SSL e l'apertura della porta 443 e, infine, se si vogliono prevenire gli attacchi di sicurezza come clickjacking e MIME-type sniffing, sono ancora necessarie le configurazioni a livello di header che possono essere fatte direttamente nel file `httpd.conf`. Per questi e altri scenari più avanzati, rimandiamo alla documentazione ufficiale su: `http://aspit.co/bou`.

E con l'approfondimento di Apache si conclude la panoramica su come possono essere configurati due web server che, a differenza di IIS, sono in grado di funzionare su piattaforme diverse da Windows. Per chi non ha queste esigenze, affronteremo ora il tema della distribuzione su IIS.

Configurazione di IIS

Internet Information Service (IIS) è sicuramente il web server più conosciuto per chi proviene dal mondo Windows e rimane uno dei web server più sicuri al mondo. La sua complessità è notevole ma, all'incirca come già visto per Apache, la sua architettura è formata da diversi moduli componibili fra di loro. Al contrario di quanto abbiamo già visto in precedenza con Apache e NGINX, in questa parte del capitolo daremo per scontata l'installazione e la configurazione stessa di IIS poiché dipende dalla versione di Windows (sulla parte server, per esempio, è attiva di default, mentre sulla parte client va attivata tramite le funzionalità aggiuntive) e dai vari ruoli che si vogliono aggiungere, come Windows Authentication o WebSockets. Nonostante l'installazione sia già data per scontata, questo ancora non consente l'esecuzione delle applicazioni ASP.NET Core out-of-the-box perché .NET Core è un framework relativamente nuovo e IIS non ha le informazioni necessarie per sapere come comportarsi: per questo è necessario procedere all'installazione del .NET Core Windows Server Hosting bundle, ovvero di un pacchetto che contiene runtime e librerie relative al framework, in aggiunta al modulo AspNetCoreModule di IIS, che è proprio la componente che sa come comunicare con l'applicazione. Il download di questo bundle è possibile all'url: `http://aspit.co/boy`. Una volta installato e riavviata la macchina, sarà visibile all'interno dei moduli utilizzabili dal web server, come mostrato nella Figura 22.7.

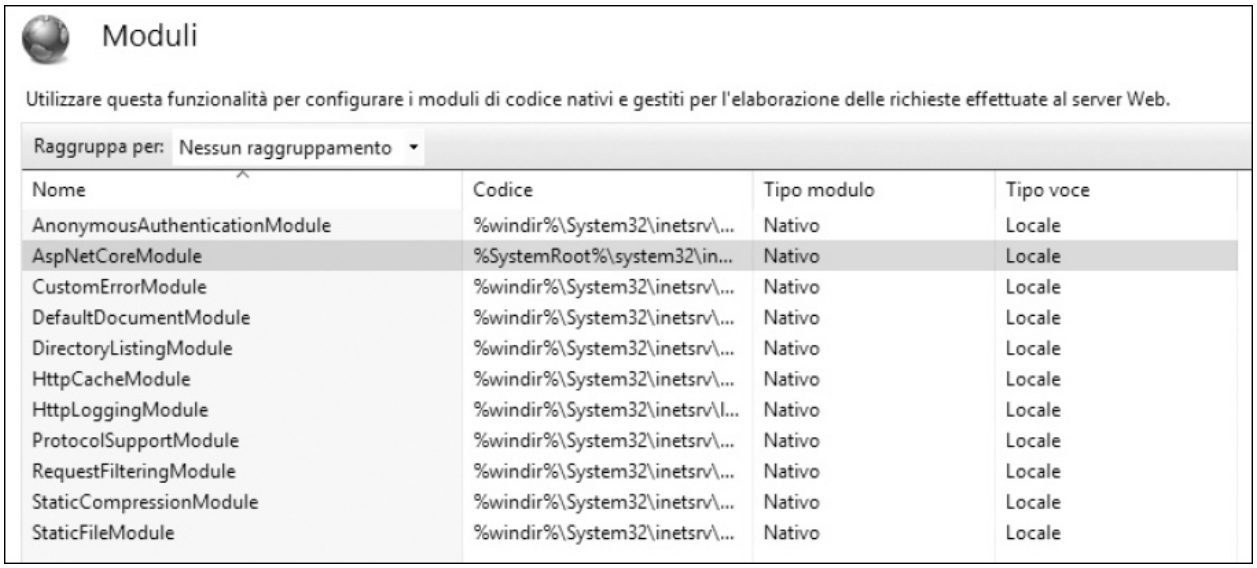


Figura 22.7 – Il modulo `AspNetCoreModule` è visibile solamente quando è stato installato correttamente il .NET Core Windows Server Hosting bundle e il runtime di ASP.NET Core.

La parte importante è che l'AspNetCoreModule non solo è in grado di eseguire le applicazioni ASP.NET Core ma permette anche la comunicazione in reverse proxy con Kestrel, senza dover cambiare una riga di codice dell'applicazione poiché il tutto viene già gestito da `IWebHostBuilder` della classe `Startup`. Inoltre, dal momento che rimanda tutto il traffico verso Kestrel, questo modulo ci permette anche di gestire scenari più avanzati relativi a sicurezza, logging e configurazione applicativa e del server stesso.

Una volta completato il setup di base, arriva il momento di creare la configurazione applicativa: ogni applicazione per girare ha bisogno di un suo application pool, ovvero di un sistema che garantisca isolamento tra le applicazioni presenti sullo stesso server, così che se una genera errori oppure va in crash, non è in grado di danneggiare le altre. Si deve quindi procedere e creare un nuovo application pool, in cui la particolarità, rispetto a quanto si è abituati dal passato con le applicazioni ASP.NET, è che ASP.NET Core non ha bisogno del CLR per partire e pertanto, come dimostra la Figura 22.8, si può impostare sul valore "No managed code".

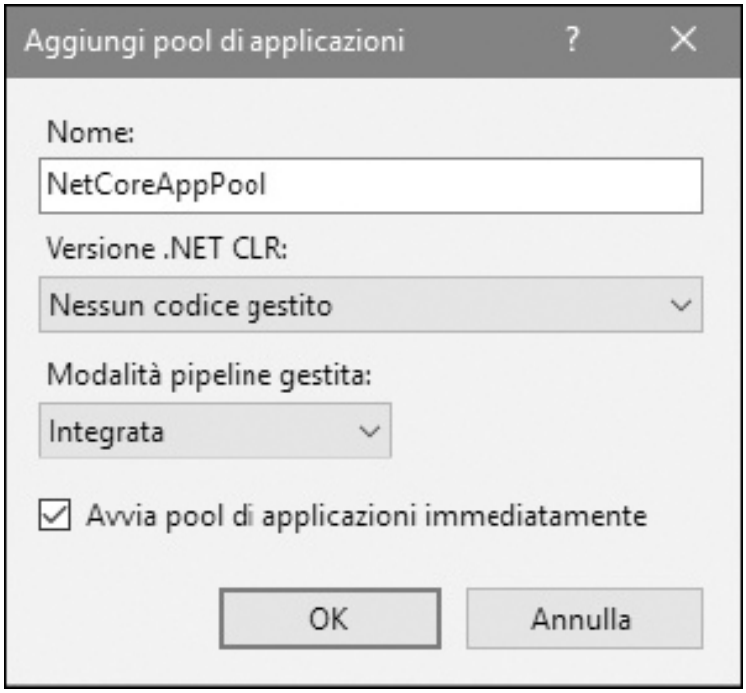


Figura 22.8 – Il nuovo application pool si può configurare con click destro sul nodo degli application pool e va impostato in modalità "No managed code" poiché .NET Core non richiede il Desktop CLR.

Una volta definito l'application pool, è possibile andare ad aggiungere il sito web, cliccando con il tasto destro sul nodo "Sites" e creandone uno nuovo. Alcuni dei parametri sono esposti nella Figura 22.9, mostrata qui di seguito.

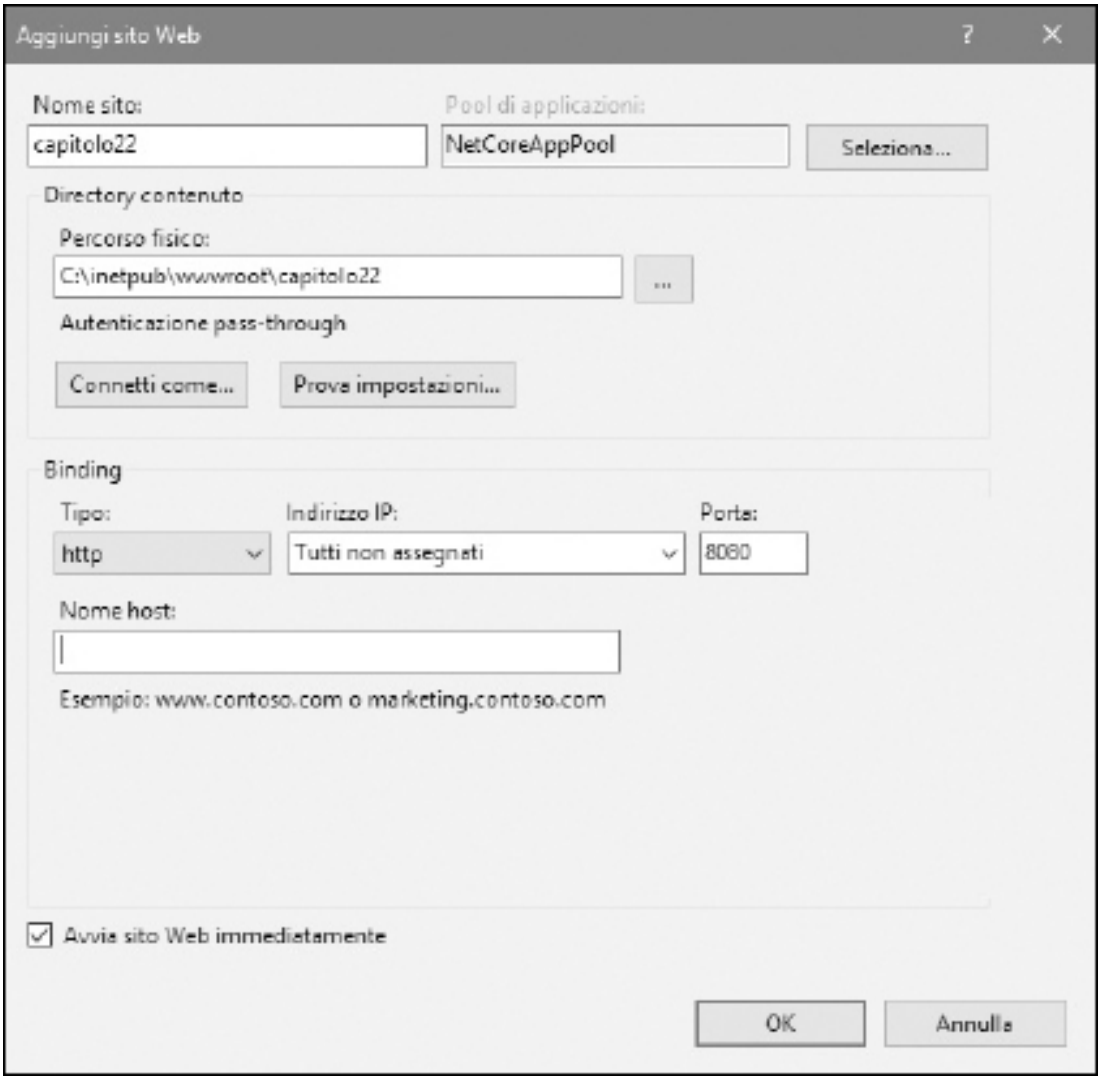


Figura 22.9 – Configurazione di nuovo sito web all'interno di IIS per ospitare l'applicazione ASP.NET Core.

Dalla Figura 22.9 si possono notare alcune caratteristiche: il nome del sito è rilevante solamente a livello dell'IIS, il nome dell'host, al contrario, è rilevante quando lo si vuole esporre all'esterno tramite certificati SSL e domini personalizzati, l'application pool selezionato è quello impostato nel passaggio precedente e, infine, il percorso fisico corrisponde alla cartella in cui è stato lanciato il comando `dotnet publish`. Andando ad analizzare l'output generato dalla pubblicazione di un'applicazione ASP.NET Core, troviamo, tra tutti, un file fondamentale per l'avvio dell'applicazione, ovvero il web.config mostrato nell'Esempio 22.16, che serve a specificare all'IIS tutti i parametri relativi all'applicazione e a IIS stesso.

Esempio 22.16

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
   <system.webServer>
      <handlers>
         <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
         resourceType="Unspecified" />
      </handlers>
      <aspNetCore processPath="dotnet" arguments=".\Capitolo22.dll" stdoutLogEnabled="false"
                  stdoutLogFile=".\logs\stdout" />
   </system.webServer>
</configuration>
```

Nell'Esempio 22.16, infatti, è evidente che nel nodo relativo al web server (IIS) si sta specificando che deve essere caricato il modulo `AspNetCoreModule`, proprio quello che è stato installato con il bundle: è grazie a questo che IIS, partendo, sarà in grado di capire che deve caricare il modulo corretto, e quindi avviare i comandi definiti dagli attributi `processPath` e `arguments` del nodo `aspNetCore`, che rappresentano, guarda caso, i comandi di avvio dell'applicazione stessa. La generazione del file `web.config` è

automatica sui progetti che hanno impostato come target di progetto `Microsoft.NET.Sdk.Web` e in cui non è già presente un file che sia personalizzato. Questo passaggio è obbligatorio e fondamentale, perché senza di lui l'applicazione stessa non sarebbe in grado di partire e pertanto questo file di configurazione non va mai rimosso. Se tutti i passaggi elencati sono stati eseguiti correttamente, navigando all'indirizzo `http://localhost:8080` (o nella porta definita in fase di setup), saremo nuovamente in grado di visualizzare l'applicazione creata.

La pubblicazione all'interno dei web server che abbiamo affrontato finora è piuttosto semplice ma richiede ancora una cosa che, nel mondo moderno, non è detto che sia più disponibile: il server fisico. Sempre più spesso, infatti, si sta cercando di spostare tutti i workload in ambienti cloud, in modo da risparmiare quanto più possibile in termini di costi e tempi di rilascio, ma garantire al tempo stesso una maggiore scalabilità sia orizzontale sia verticale, praticamente immediata, di migliaia di istanze che a oggi non è possibile in data center on-premise. Proprio per via di queste premesse, discuteremo ora di com'è possibile portare la stessa applicazione web anche in un ambiente cloud pubblico di Microsoft Azure e ne illustreremo i servizi gestiti.

Pubblicazione con Microsoft Azure

Microsoft Azure offre, come gli altri cloud vendor, la possibilità di creare servizi e scalarli potenzialmente all'infinito, secondo le proprie esigenze, con una novità per quanto riguarda la fatturazione: al contrario di quanto avviene on-premise, in cui il costo dell'hardware e di eventuali servizi necessari alle applicazioni devono essere sostenuti a priori, con l'avvento del cloud i costi sono limitati alle risorse che vengono effettivamente utilizzate. Entrando nel mondo delle applicazioni web, abbiamo già visto come sia possibile configurare un proprio web server e fare l'hosting. Pertanto, se si volesse andare nel cloud, il primo passo sarebbe di utilizzare il servizio delle macchine virtuali. Il problema nell'adottare una soluzione di questo tipo è solitamente relativo ai costi che si riflettono su diversi aspetti: manutenzione del sistema operativo, gestione delle patch, configurazione del web server, configurazione della scalabilità delle macchine virtuali, gestione del cluster per mantenere alta l'affidabilità, creazione del load balancing e così via. Analizzando meglio questi punti, è abbastanza evidente che non si trae alcun vantaggio nell'andare nel cloud con una soluzione del genere, perché ci si porterebbe dietro gli stessi problemi dell'on-premise, meno la gestione dell'hardware, e per questo conviene adottare quelli che vengono definiti servizi gestiti (PaaS), in cui tutto l'hardware e il servizio stesso vengono gestiti direttamente da Microsoft, lasciandoci solamente possibilità di personalizzazioni sull'ambiente ma senza avere l'accesso alle macchine. Il tutto si traduce in una serie di vantaggi e di riduzioni dei costi, sia da parte del cloud, perché sono coinvolte meno risorse, sia da parte delle aziende, perché sono necessarie meno persone specializzate sull'infrastruttura.

Il servizio gestito in Microsoft Azure, che si occupa dell'hosting delle applicazioni web, si chiama App Service e per utilizzarlo è richiesta solamente l'attivazione di una sottoscrizione (anche gratuita) che può essere fatta da questo link: `http://aspit.co/boz`. Una volta attivato l'account, tutto il resto delle operazioni può essere fatto direttamente all'interno di Visual Studio tramite una serie di menù guidati: il primo passo è quello di cliccare con il tasto destro sul progetto che si vuole pubblicare e quindi scegliere l'opzione "Publish". Questo mostrerà a schermo un menù come quello evidenziato nella Figura 22.10.

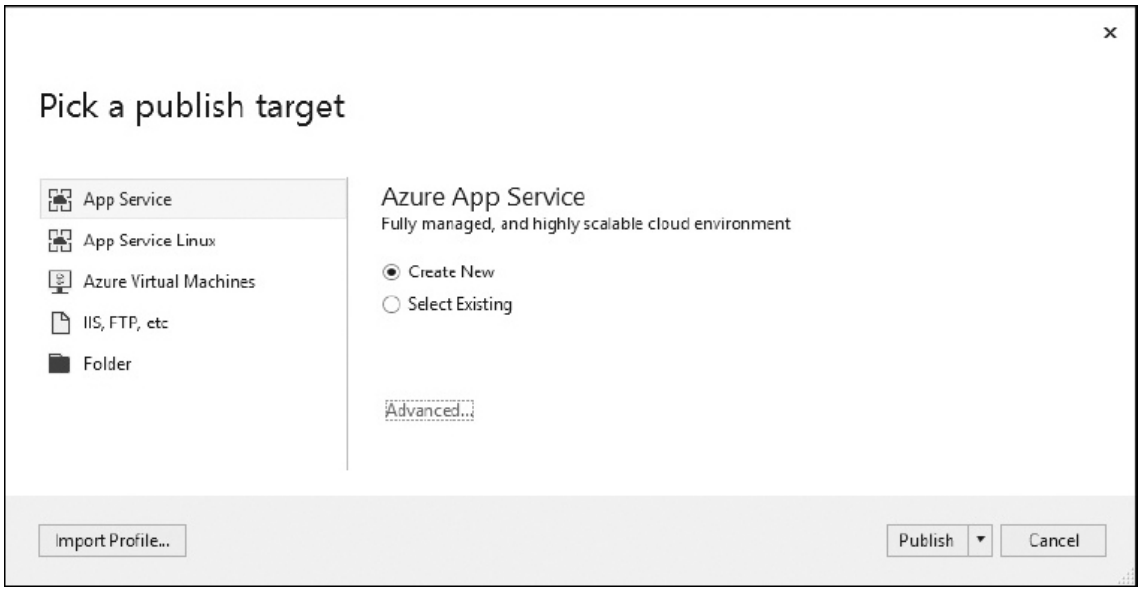


Figura 22.10 – Il menù di pubblicazione di Visual Studio offre diversi target, tra cui gli Azure App Service (Windows o Linux), macchine virtuali, IIS/FTP e cartelle definite in locale.

Come si può notare dalla Figura 22.10, sono disponibili diverse opzioni, tra cui la stessa macchina virtuale in cui si dovrà configurare il web server, piuttosto che la pubblicazione tramite IIS, FTP o una cartella, che non fanno altro che richiamare il comando `dotnet publish`

in un percorso specifico. Tra le opzioni relative al cloud, invece, ci sono due livelli di App Service perché, dato che si sta sviluppando un'applicazione ASP.NET Core, si potrebbe voler distribuire l'applicazione non solo su Windows ma anche su ambiente Linux. Discuteremo successivamente di come questo venga fatto, parlando dei Docker container. Prima di creare il servizio, dato che al momento da portale non è stato ancora creato, è necessario assicurarsi che le impostazioni di pubblicazione dell'applicazione stessa siano corrette. Pertanto, si può cliccare sul pulsante "Advanced" ed effettuare le modifiche, come illustrato nella Figura 22.11.

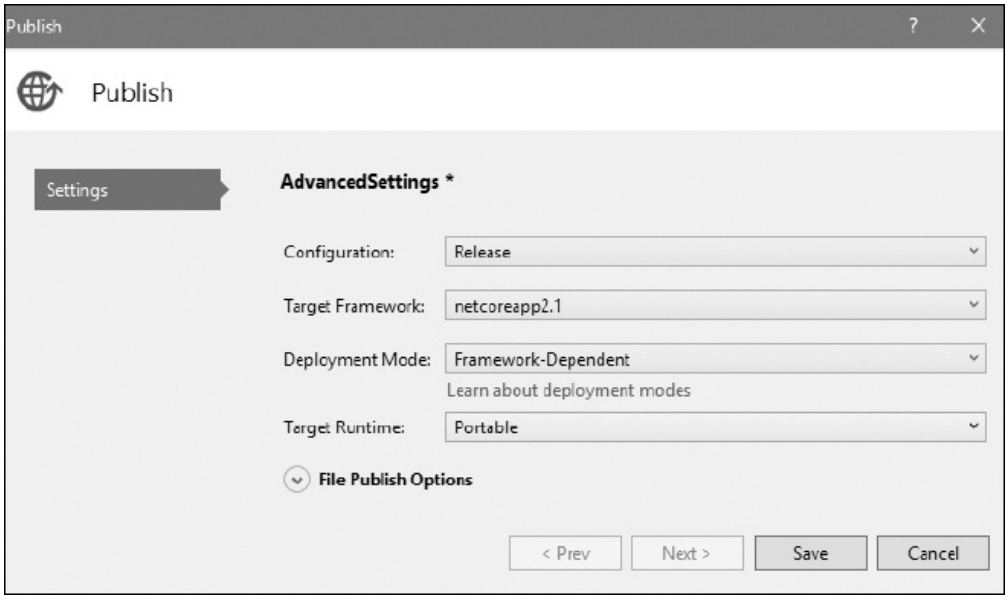


Figura 22.11 – Ogni progetto ASP.NET Core può avere diverse modalità di pubblicazione (framework-dependent o self-contained).

Il servizio degli App Service prevede già un'installazione di .NET Core, perciò si può sfruttare una tipologia di deployment framework-dependent, come mostrato nella Figura 22.11, ma in caso in cui ci sia l'esigenza di un altro runtime, si può comunque fare il deployment di tipo self-contained. Una volta salvate le impostazioni di pubblicazione e cliccato il pulsante "Publish" del menù della Figura 22.10, verrà richiesto di fare il login con l'account Microsoft con il quale è stata creata la sottoscrizione di Azure e quindi verrà mostrato il menù della Figura 22.12, in cui sarà possibile creare la nuova risorsa.

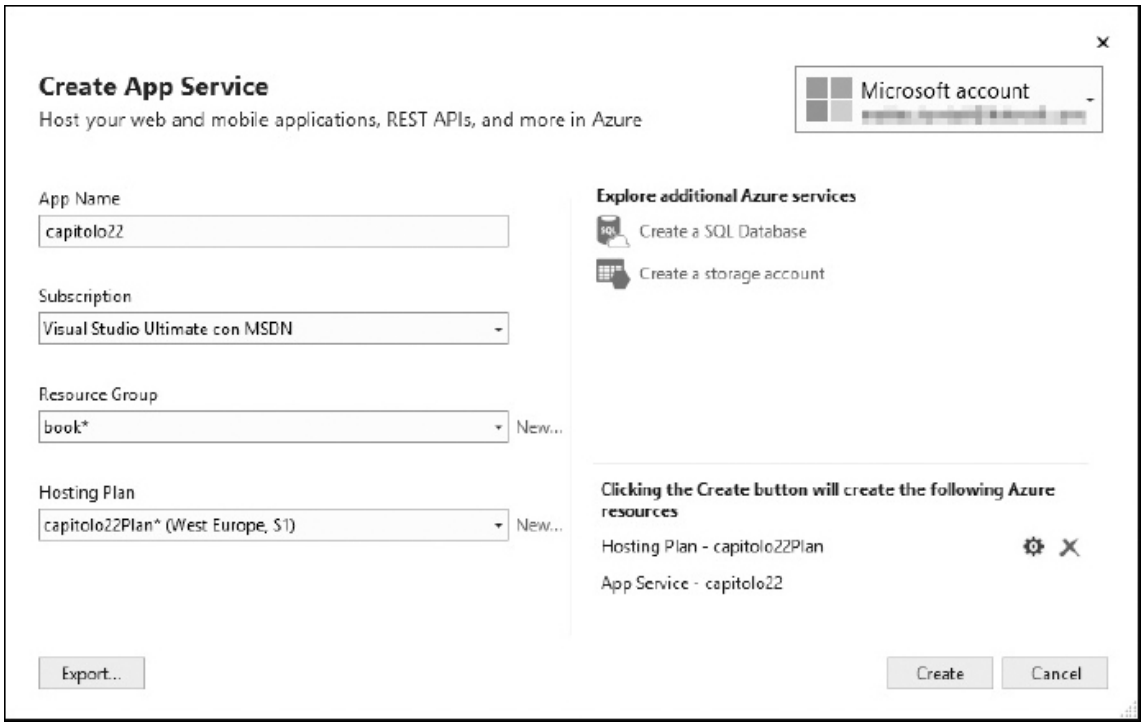


Figura 22.12 – Il menù di creazione di una nuova istanza di Azure App Service permette di aggiungere anche servizi esterni, come un Azure SQL Database o un Azure Storage Account per salvare dati.

La creazione di una nuova risorsa di tipo App Service richiede quattro parametri:

- App Name: rappresenta il nome DNS che avrà l'applicazione, perciò deve essere unico in tutto il mondo. L'URL completo dell'applicazione sarà:`https://{AppName}.azurewebsites.net` di default, ma sarà possibile cambiarlo tramite portale, con l'aggiunta di certificati SSL e domini personalizzati.

- Subscription: la sottoscrizione attiva verso la quale verrà effettuata la fatturazione mensile.

- Resource Group: permette di rappresentare un elenco di risorse che sono correlate fra di loro (come, per esempio, un sito web e il relativo database) in un gruppo in cui sono tutte visibili.

- Hosting plan: indica il piano di servizio, ovvero la grandezza della macchina di una singola istanza che eseguirà l'applicazione e la sua posizione geografica.

La configurazione del piano di servizio viene fatta tramite un menù secondario, come è visibile nella Figura 22.13.

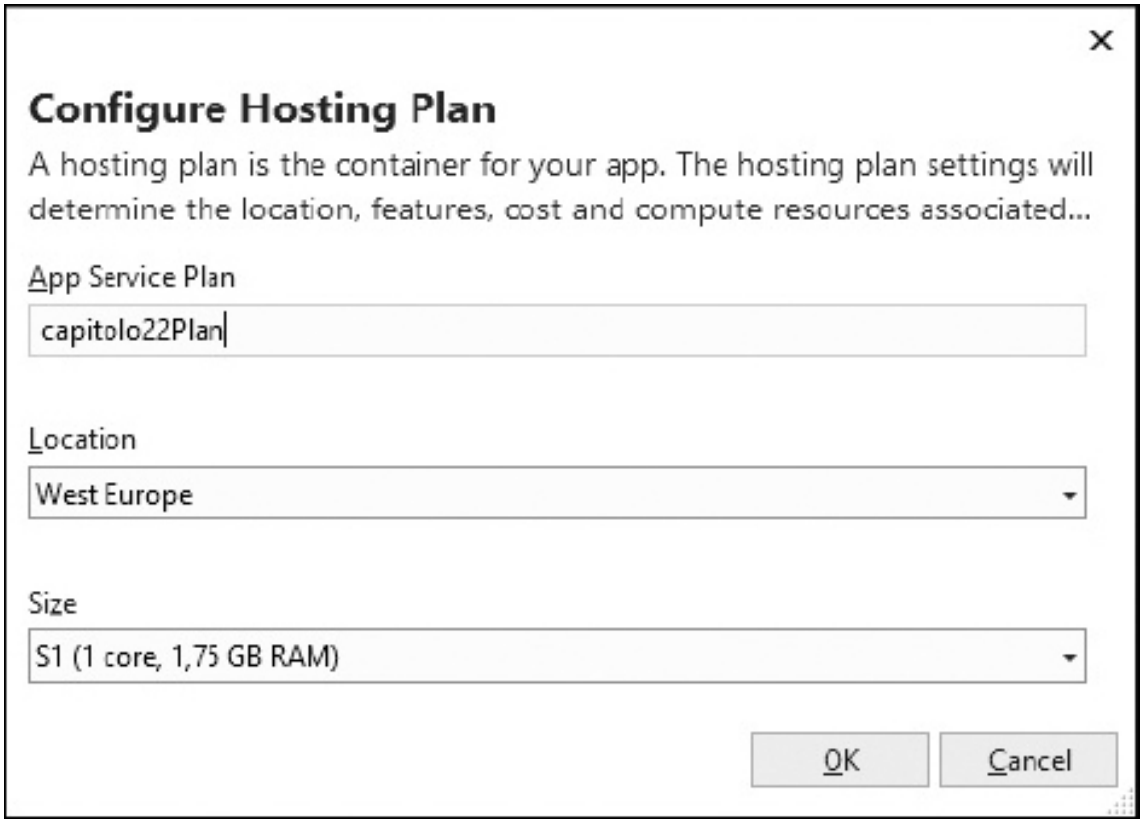


Figura 22.13 – La configurazione del piano di servizio include la scelta del data center e della dimensione della singola istanza dell'App Service.

Una volta configurata la risorsa base per l'hosting dell'applicazione web, è eventualmente possibile configurare il rilascio anche delle risorse collegate come, per esempio, il database. La procedura sarà molto simile, però, su Microsoft Azure, verrà creata, per esempio, una risorsa gestita di tipo SQL Database, che permetterà di gestire i dati relazionali come se fosse un vero e proprio SQL Server. Cliccando sul pulsante "Create" del menù mostrato nella Figura 22.12, verranno eseguite due operazioni: la prima è la creazione, all'interno della cartella "Properties" del progetto, di un file chiamato `{AppName}-WebDeploy.pubxml` contenente tutta la configurazione appena effettuata, mentre la seconda è la pubblicazione all'interno del servizio cloud. Quando l'operazione di pubblicazione si concluderà, saremo reindirizzati dal browser al vero e proprio sito web appena creato, come risulta visibile nella Figura 22.14.

Le pubblicazioni che verranno eseguite successivamente, per via di modifiche all'interno del codice, saranno molto più rapide perché la configurazione è già salvata all'interno del publishing profile e, per via del meccanismo di deployment scelto, verranno pubblicate solamente le modifiche e non tutta l'applicazione.

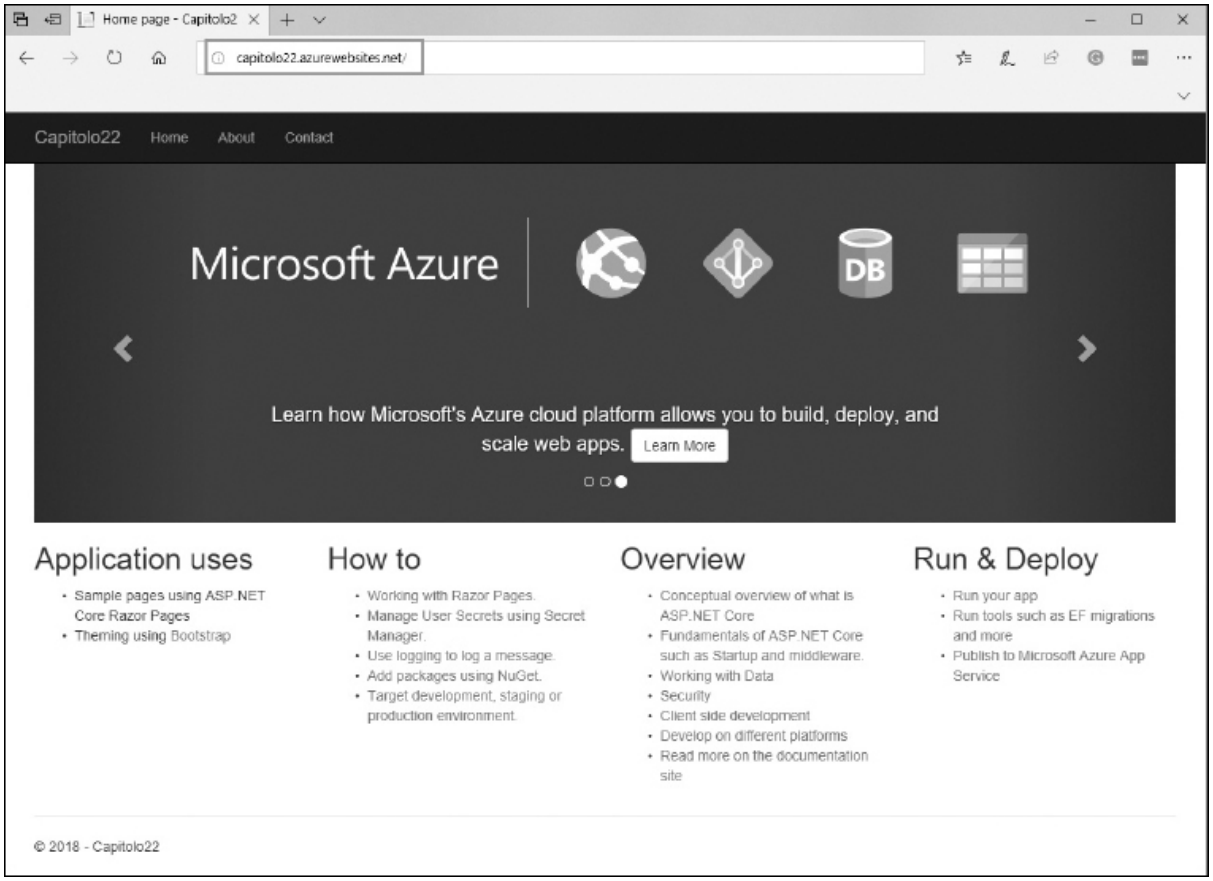


Figura 22.14 – L'applicazione distribuita su App Service è raggiungibile tramite un URL pubblico.

Accedendo al portale su `https://portal.azure.com` sarà possibile modificare alcune delle impostazioni del servizio creato, tra cui, per esempio, l'attivazione delle WebSocket per SignalR, piuttosto che HTTP/2 o, ancora, l'impostazione delle variabili d'ambiente come `ASPNETCORE_ENVIRONMENT` per modificarne la tipologia di esecuzione al volo, come mostrato nella Figura 22.15.

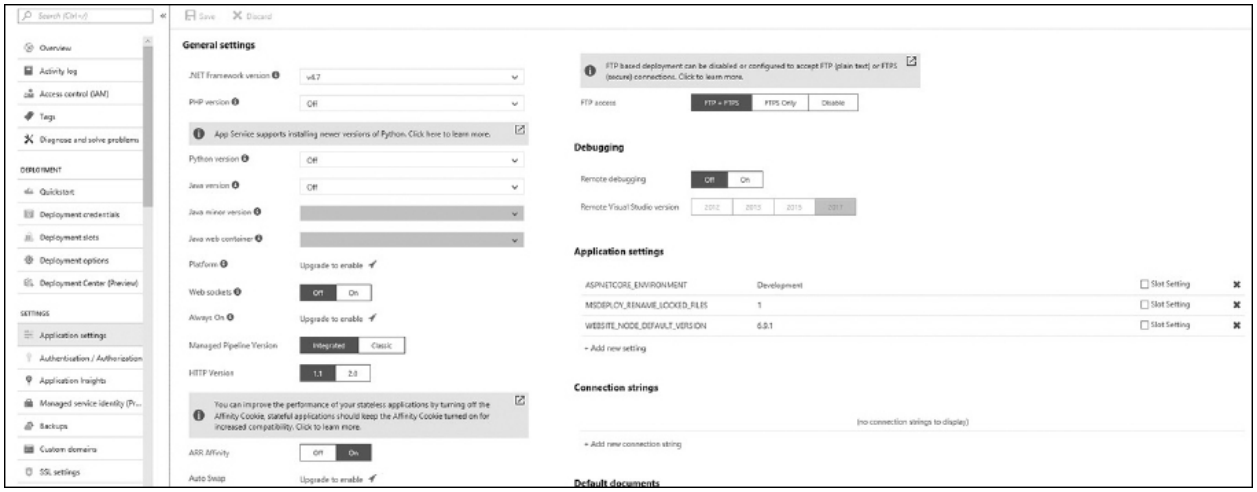


Figura 22.15 – All'interno dell'App Service specificato, si possono cambiare le impostazioni relative all'applicazione stessa ma anche all'ambiente sulla quale viene eseguita.

Un aspetto particolarmente interessante degli App Service è che offrono la possibilità di attaccare il debugger di Visual Studio per determinare, per esempio, come mai alcuni problemi si verificano solamente una volta pubblicata l'applicazione. Questa funzionalità si chiama Remote Debugging e va attivata in modo esplicito, selezionando opportunamente la versione di Visual Studio che si desidera utilizzare direttamente all'interno del portale di Azure, ed è inoltre necessario che l'applicazione distribuita nell'App Service sia stata compilata in modalità di debug. All'interno di Visual Studio, aprendo la finestra del Server Explorer, sarà possibile scegliere tra i nodi Azure -> App Service -> Resource Group l'applicazione che si deve debuggare e, cliccando con il tasto destro del mouse sopra di essa, selezionare l'opzione "Attach Debugger", come mostrato nella Figura 22.16.

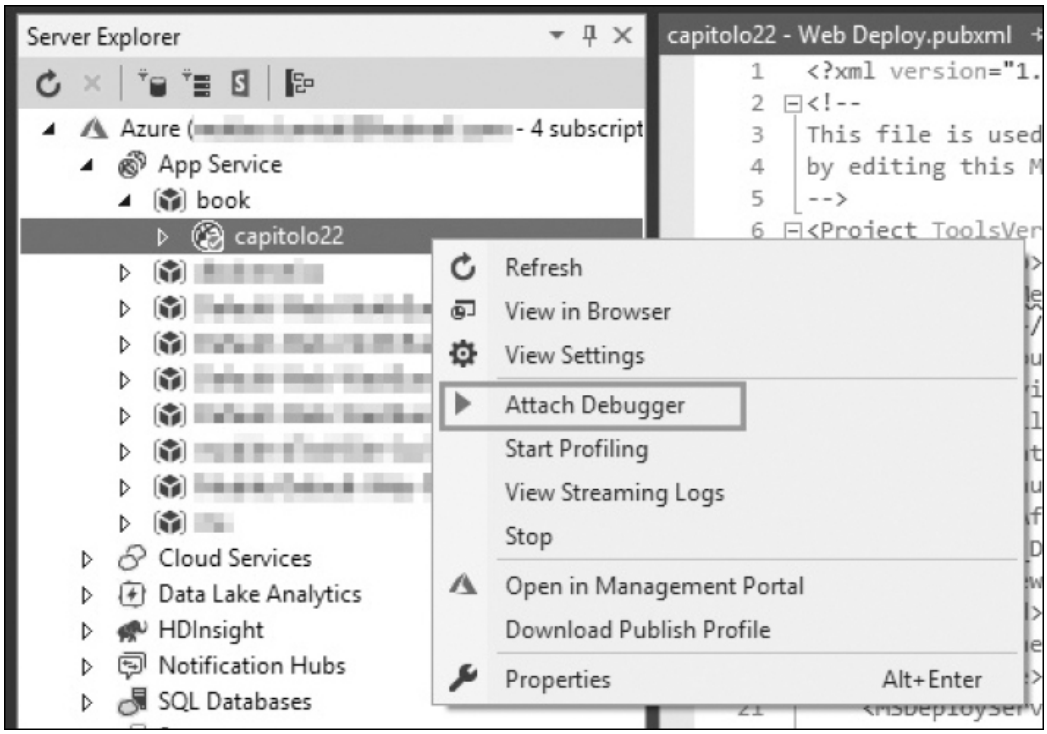


Figura 22.16 – L'opzione "Attach Debugger" è disponibile tramite il menù secondario dal Server Explorer sull'istanza di App Service selezionata.

Questa funzionalità è estremamente potente e non deve essere utilizzata in scenari di produzione: ogni qualvolta si incontri, per esempio, un breakpoint all'interno di una action tramite una route specificata dall'utente, tutta l'esecuzione dell'applicazione verrà bloccata fino a quando non verrà eseguito lo step successivo ed eventualmente rilasciato il debugger. Testare in remoto è comunque molto utile perché, essendo il servizio gestito, non è possibile essere a conoscenza dei dettagli implementativi dei server che offrono il servizio stesso; pertanto, attivando il debugger, si potrebbero scoprire situazioni non verificabili e non riproducibili nell'ambiente di sviluppo o di staging.

Gli Azure App Service includono molte altre funzionalità, fra cui la personalizzazione dei domini, l'autenticazione, la gestione delle performance e logging tramite Application Insights, o la scalabilità automatica ma, poiché non interessano il tema centrale del libro, rimandiamo alla documentazione ufficiale su http://aspit.co/bo0 per approfondire le potenzialità di questi servizi gestiti.

In generale, soluzioni come quelle che abbiamo affrontato all'interno di questo capitolo funzionano, ma non sono l'ideale nei casi in cui si stia sviluppando un sistema orientato ai micro-servizi: in questi casi specifici, infatti, distribuire e orchestrare tutte le versioni, piuttosto che gestire le dipendenze fra i vari servizi, diventa via via più complesso, pertanto c'è bisogno di un sistema che semplifichi ulteriormente il rilascio.

Pubblicazione con Docker

Uno dei problemi più sentiti da parte degli sviluppatori è che non è possibile lavorare con ambienti completamente isolati e replicabili. Infatti, spesso si sente parlare del fenomeno "it works on my machine" (letteralmente "funziona sul mio PC") per via del fatto che l'installazione di una nuova dipendenza, se non comunicata agli altri membri del team di sviluppo o configurata correttamente, potrebbe portare a una instabilità del sistema fino al non funzionamento dello stesso, cosa che in ambienti di produzione non dovrebbe succedere. Mettendosi invece nei panni del team di operation che deve gestire la messa in produzione delle varie applicazioni, il tutto diventa ancora più complicato perché spesso non si hanno gli script giusti per configurare correttamente gli ambienti. Inoltre, per garantire l'affidabilità e la disponibilità del servizio, si finisce spesso con il creare nuove macchine virtuali, o fisiche, per fare repliche e hosting di una sola applicazione, e questo implica che non solo ci saranno da gestire gli aggiornamenti applicativi ma bisognerà anche controllare gli aggiornamenti relativi al sistema operativo, installare patch di sicurezza, riconfigurare il web server allo stesso modo su tutte le macchine e così via. In ambienti in cui ci sono decine e decine di server dedicati, questo è un problema: non è sempre possibile avere script che ricreano la stessa configurazione su tutte le macchine, fare un deployment richiede tanto tempo in relazione al numero dei server e, oltre a questo, ci si ritrova con tanto hardware che in realtà non fa nulla. Proprio per rimediare a questi problemi, si sviluppa il mondo dei container, capeggiato principalmente da aziende come Docker, il cui scopo è proprio quello di avere la virtualizzazione di una sola porzione della macchina virtuale, così da risparmiare sui costi, mantenere alta l'efficienza senza dover gestire il sistema operativo, creando un sistema isolato, il container, in grado di tenere al suo interno tutte le dipendenze, framework e configurazioni di cui il sistema stesso ha bisogno per funzionare.

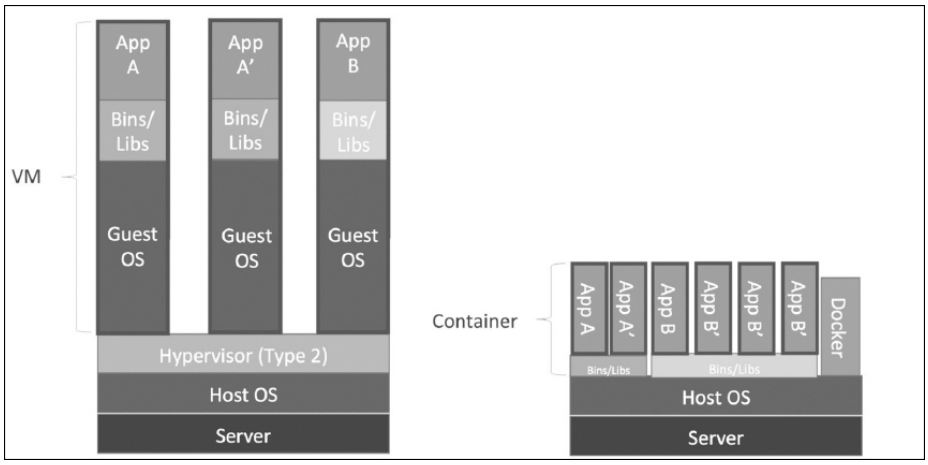


Figura 22.17 – Su una macchina virtuale c'è isolamento tra le applicazioni ma due istanze della stessa applicazione non possono condividere delle librerie in comune, perciò sono molto grandi, costose, inefficienti e complesse da gestire. I container, invece, vivono in isolamento e possono condividere le librerie attraverso un host (in questo caso il Docker Host), senza bisogno di un sistema operativo intermedio come nelle machine virtuali.

Come è dimostrato nella Figura 22.17, i container, non avendo più il sistema operativo delle VM da gestire e avendo la condivisione delle risorse e delle dipendenze, portano a una serie di vantaggi:

- Controllo granulare: l'approccio è orientato ai micro-servizi, anche se può funzionare con qualsiasi tipologia di applicazione.
- Testing facilitato: il testing (e il funzionamento) di un'applicazione in locale su container produrrà gli stessi risultati che in un qualsiasi altro ambiente, compresa la produzione.
- Deployment semplificato: l'applicazione è pacchettizzata con tutte le sue dipendenze in un solo componente, ovvero il container. In caso di rilascio di una nuova versione, i container possono girare in parallelo, consentendo anche la creazione di scenari di A/B testing.
- Avvio rapido: meno di un secondo, in genere, poiché c'è cache sulle immagini che creano i container.

Considerati tutti i vantaggi di questa nuova tipologia di sviluppo e deployment, Microsoft ha deciso di approcciarla facendo un'integrazione più semplice possibile direttamente all'interno di Visual Studio per i progetti .NET Core e ASP.NET Core. Nonostante cambi tutta la tecnologia, la semplicità nasce dal fatto che, tramite Visual Studio, il modello di sviluppo non cambia e si continuerà a lavorare come è stato sempre fatto, ma facendo il click con il tasto destro del mouse e selezionando Add -> Docker Support, come mostrato nella Figura 22.18, si avranno tutti i vantaggi illustrati nel paragrafo precedente.

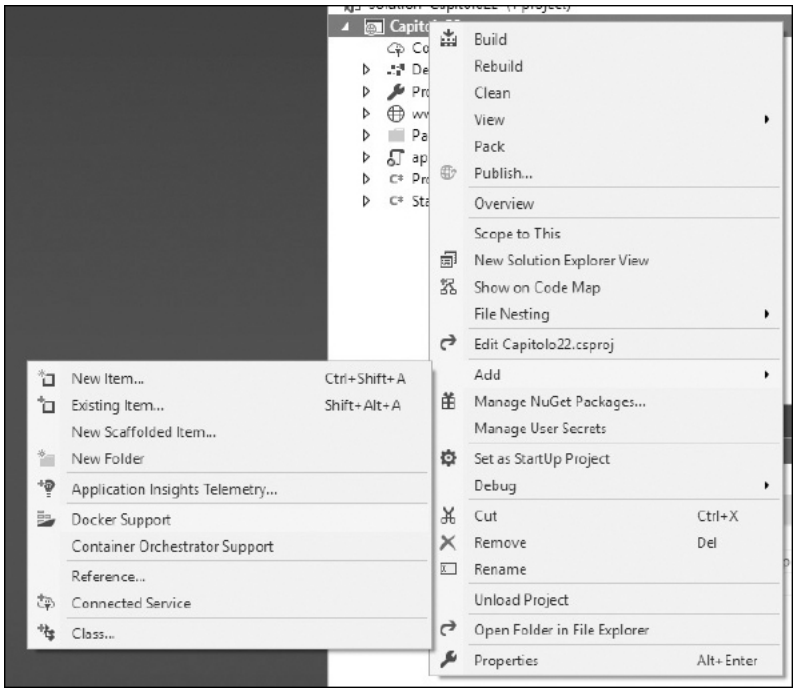


Figura 22.18 – L'aggiunta del supporto a Docker è fatta tramite il tasto destro sul progetto che si vuole containerizzare e selezionando l'opzione Add -> Docker Support

.

Quello che avviene dopo questa operazione, mostrata nella Figura 22.18, è la creazione di un progetto, chiamato `docker-compose` a livello di solution, e di un file, chiamato `Dockerfile`, all'interno del progetto selezionato. Nella solution è stato creato un nuovo progetto, che contiene una serie di file, tra cui il `.dockerignore` che serve ai sistemi di source control per non fare il commit di alcuni tipi di file relativi a Docker, e il `docker-compose.yml`, un file con una sintassi molto simile al JSON, in cui vengono descritti tutti i servizi che devono essere creati, come dimostra l'Esempio 22.17.

Esempio 22.17

```
version: '3.4'

services:
   capitolo22:
      image: capitolo22
      build:
         context: .
         dockerfile: Capitolo22/Dockerfile
```

Nell'Esempio 22.17 si può notare come il servizio, in questo caso, sia solo uno perché il progetto sul quale abbiamo abilitato questa tecnologia è solamente uno e verrà creato a partire dal file `Dockerfile` contenuto nella sua soluzione. Quello che verrà creato non è il vero e proprio container ma solo una immagine che avrà il nome specificato nel tag image, ovvero "capitolo22": il container non sarà altro che l'esecuzione dell'immagine stessa; per questo, a livello di avvio e scalabilità, è molto più efficiente.

Il `Dockerfile`, ovvero il file di configurazione che specifica a Docker come deve essere costruita l'immagine, è stato creato di default sulla versione dell'SDK che abbiamo dichiarato a livello di progetto, in questo caso la versione 2.1 di ASP.NET Core. Pertanto il risultato ottenuto in automatico da Visual Studio sarà qualcosa di simile a quello mostrato nell'Esempio 22.18.

Esempio 22.18

```
FROM microsoft/dotnet:2.1-aspnetcore-runtime AS base
WORKDIR /app
EXPOSE 80

FROM microsoft/dotnet:2.1-sdk AS build
WORKDIR /src
COPY Capitolo22/Capitolo22.csproj Capitolo22/
RUN dotnet restore Capitolo22/Capitolo22.csproj
COPY . .
WORKDIR /src/Capitolo22
RUN dotnet build Capitolo22.csproj -c Release -o /app

FROM build AS publish
RUN dotnet publish Capitolo22.csproj -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Capitolo22.dll"]
```

Nell'Esempio 22.18 viene mostrata una funzionalità particolare di Docker, chiamata multi-staged build e, infatti, si stanno susseguendo quattro fasi distinte dalla parola chiave `FROM`, che identifica un nuovo gruppo di operazioni:

- Fase 1: viene costruita una prima immagine, temporanea e nominata base, a partire da un'immagine già esistente e pre-configurata chiamata `dotnet`, che contiene il runtime di ASP.NET Core 2.1. Sebbene non sia obbligatorio, è molto utile per non partire dall'immagine vuota di Docker chiamata `SCRATCH`, in cui bisognerebbe lanciare comandi PowerShell o Bash in sequenza per installare tutto il runtime. Successivamente, viene creata una cartella chiamata app e quindi viene aperta la porta 80, che servirà a comunicare con l'esterno del container.

- Fase 2: viene costruita una seconda immagine, temporanea e nominata `build`, a partire da un'immagine già esistente chiamata sempre `dotnet`, che però questa volta contiene l'SDK di .NET Core 2.1, quindi vengono copiati i file di progetto e tutto il codice sorgente e, una volta impostata la working directory nella cartella corretta, viene lanciato il comando `dotnet build` con la configurazione di `Release`.

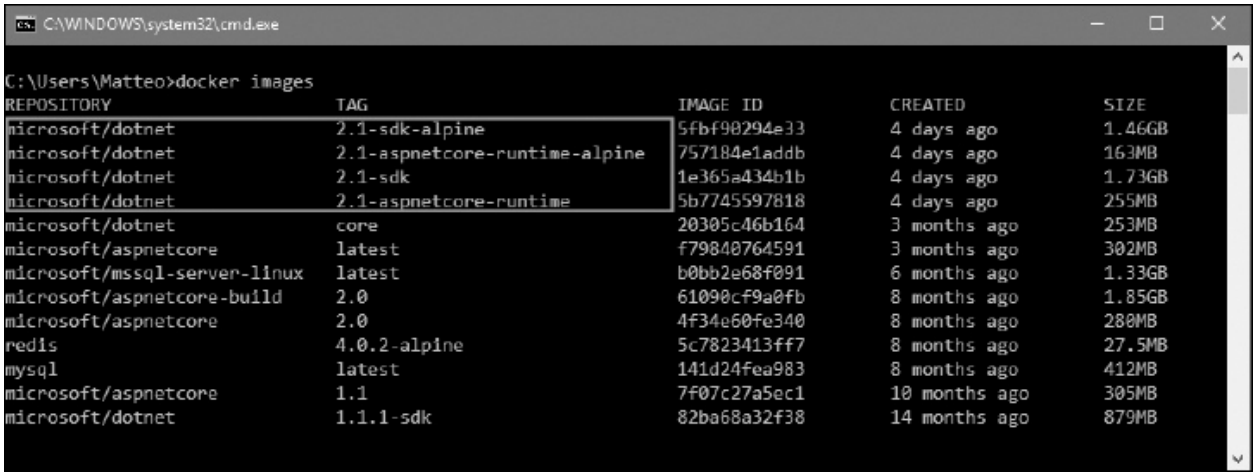
- Fase 3: viene costruita una terza immagine, sempre temporanea e nominata `publish`, a partire dall'immagine `build` creata nella fase 2, in cui avviene la pubblicazione tramite il comando `dotnet publish` in configurazione di `Release`, in cui viene anche specificato il percorso della cartella di output. Questo passaggio poteva anche essere inglobato con l'immagine generata in precedenza, ma per avere una maggiore separazione tra le varie operazioni è stata suddivisa in un'immagine a parte. Il

comando `dotnet publish` è possibile perché, dato che l'immagine di build è accessibile, lo saranno anche i sorgenti e i vari file di progetto in essa contenuti.

- Fase 4: viene costruita una quarta e ultima immagine a partire dall'immagine base costruita sul runtime nella Fase 1, in cui avviene la copia dei soli file contenuti nella cartella di pubblicazione dell'immagine costruita in Fase 3, così da non avere più i sorgenti ma solo le DLL compilate, quindi viene impostato l'entrypoint che sarà corrispondente al comando `dotnet Capitolo22.dll` che farà partire a tutti gli effetti l'applicazione ASP.NET Core.

Come possiamo notare, le immagini base dalla quale si è partiti sono due: il runtime e l'SDK. Poiché tutti i processi di build e pubblicazione vengono eseguiti direttamente all'interno del container, data la definizione del `Dockerfile`, c'è bisogno di un'immagine con tutto l'SDK per effettuare queste due operazioni, mentre quando si deve eseguire l'applicazione è sufficiente il runtime, che permette anche di risparmiare diverse centinaia di megabyte di spazio. L'operazione di build e pubblicazione all'interno del container stesso non è molto utile quando si lavora in un ambiente in cui c'è a disposizione Visual Studio ma è fondamentale in ambienti in cui non c'è installato .NET Core. È possibile vedere le immagini scaricate e la loro dimensione su disco lanciando il comando `docker images`, come dimostra la Figura 22.19.

L'immagine generata partendo dal `Dockerfile` dell'Esempio 22.19 sarà, come visibile nella Figura 22.19, di circa 250Mb, che, considerando un ambiente Linux, non è poco: ogni volta che si deve rilasciare un aggiornamento, viene potenzialmente spostata tutta l'immagine (anche se non è del tutto vero, dato che viene verificata la cache sui vari strati di cui è composta), quindi, più sarà grande, più tempo ci metterà a essere trasferita via network, più banda occuperà e più tempo ci metterà a partire in caso di scalabilità. Una delle novità introdotte a partire da .NET Core 2.1 riguarda l'introduzione di nuove immagini basate su Alpine, ovvero una variante di Debian, pensata per essere leggera. Infatti, l'immagine contenente il solo runtime arriva a pesare poco più di 80Mb, che rappresenta una differenza di circa 170Mb rispetto al passato. Per utilizzarla, sarà sufficiente modificare le istruzioni `FROM` dell'Esempio 22.18, impostando come immagini base `microsoft/dotnet:2.1-aspnetcore-runtime-alpine` per il runtime e `microsoft/dotnet:2.1-sdk-alpine` per l'sdk. Tutte le immagini che possono essere utilizzate per runtime e SDK sono visibili all'interno del Docker Hub di Microsoft, ovvero di un repository in cui sono salvate tutte le immagini pre-costruite e pronte da utilizzare, a



```
C:\WINDOWS\system32\cmd.exe

C:\Users\Matteo>docker images
REPOSITORY                  TAG                         IMAGE ID        CREATED         SIZE
microsoft/dotnet            2.1-sdk-alpine              5fbf90294e33    4 days ago      1.46GB
microsoft/dotnet            2.1-aspnetcore-runtime-alpine 757184e1addb  4 days ago      163MB
microsoft/dotnet            2.1-sdk                     1e365a434b1b    4 days ago      1.73GB
microsoft/dotnet            2.1-aspnetcore-runtime      5b7745597818    4 days ago      255MB
microsoft/dotnet            core                        20305c46b164    3 months ago    253MB
microsoft/aspnetcore        latest                      f79840764591    3 months ago    302MB
microsoft/mssql-server-linux latest                     b0bb2e68f091    6 months ago    1.33GB
microsoft/aspnetcore-build  2.0                         61090cf9a0fb    8 months ago    1.85GB
microsoft/aspnetcore        2.0                         4f34e60fe340    8 months ago    280MB
redis                       4.0.2-alpine                5c7823413ff7    8 months ago    27.5MB
mysql                       latest                      141d24fea983    8 months ago    412MB
microsoft/aspnetcore        1.1                         7f07c27a5ec1    10 months ago   305MB
microsoft/dotnet            1.1.1-sdk                   82ba68a32f38    14 months ago   879MB
```

partire da questo indirizzo: http://aspit.co/bo2.

Figura 22.19 – Ogni immagine elencata dal comando `docker images` è composta da un nome, da un tag che ne specifica la versione, da un identificativo che la referenzia in modo univoco, da una data di creazione e da una dimensione.

Una volta dichiarato come deve essere costruita l'immagine che conterrà quindi l'applicazione e specificate le varie ottimizzazioni come l'uso di Alpine, verrà il momento di eseguirla. Osservando bene Visual Studio, dopo l'aggiunta del supporto a Docker, si noterà come è stato cambiato il progetto di startup, che non sarà più l'applicazione web stessa ma l'altro progetto `docker-compose` aggiunto da Visual Studio. Questo consente a Visual Studio di interagire con Docker e creare le immagini specificate nei `Dockerfile` a partire dalle specifiche dichiarate nel `docker-compose.yml` ma non solo: se in Docker è attiva la condivisione del disco, Visual Studio aprirà anche una porta per il debugger e, quindi, sarà possibile testare le applicazioni all'interno dei container stessi che di default sarebbero isolati. Qualora invece si volesse procedere manualmente, sarà necessario lanciare solo due comandi, come viene mostrato nell'Esempio 22.19.

Esempio 22.19

```
docker build -t capitolo22 -f Capitolo22/Dockerfile .
docker run capitolo22
```

Come dimostrato nell'Esempio 22.19, infatti, il primo passaggio è quello di costruire l'immagine a partire dal `Dockerfile` creato in automatico da Visual Studio, con un nome specificato dal parametro `t`, quindi, una volta creata l'immagine, si potrà effettivamente

avviare il container con il suo nome. A questo punto, l'applicazione ASP.NET Core verrà avviata con il comando definito dall'`ENTRYPOINT` del `Dockerfile` e ci sarà lo startup, come mostrato nella Figura 22.20.

```
Prompt dei comandi - docker run capitolo22

C:\Users\Matteo>docker run capitolo22
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
      No XML encryptor configured. Key {3ceb5435-22ab-483a-a2ae-08f63a3e0f5e} may be persisted
to storage in unencrypted form.
Hosting environment: Production
Content root path: /app
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

Figura 22.20 – L'avvio (manuale in questo caso) del container con l'applicazione ASP.NET Core farà partire Kestrel e aprirà la porta 80, esposta all'interno del container, per fare in modo che sia raggiungibile dall'esterno.

Una volta creata l'immagine e verificato il corretto funzionamento del container, si può procedere alla pubblicazione. Esistono diverse soluzioni che sono in grado di eseguire e gestire container, ma lo strumento più semplice è quello che abbiamo già visto in precedenza, ovvero gli App Service: la versione per Linux è infatti in grado di prendere l'immagine ed eseguirla, lanciando i comandi che abbiamo eseguito manualmente o tramite Visual Studio. Gli App Service sono comodi solo per un caso molto specifico però, ovvero il caso in cui ci siano pochi servizi avviati come container, tutti definiti all'interno del file `docker-compose.yml` e tutti stateless. Nei casi in cui ci sia bisogno di avviare container che devono mantenere dati, piuttosto che container che necessitano di migliaia di istanze, gli App Service non rappresentano la soluzione migliore e, proprio per questo, in Azure sono disponibili altri sistemi come Azure Kubernetes Service (AKS), Service Fabric o Azure Container Instances (ACI) che permettono di mantenere cluster e ne garantiscono l'alta affidabilità e alta disponibilità. Maggiori informazioni su questi servizi sono disponibili su `http://aspit.co/bo8`.

Conclusioni

In questo capitolo abbiamo parlato di come stanno evolvendo le esigenze sia dei team di sviluppo sia dei team di operation e di come stanno sempre di più convergendo verso il mondo dei DevOps, in cui le competenze sono costituite da un insieme di software e infrastruttura, ma non solo: anche le tempistiche dei rilasci e la possibilità di avere più ambienti in cui verificare il funzionamento sono sempre di più punti critici nella crescita di un qualsiasi prodotto. Proprio per risolvere questi problemi, abbiamo visto e affrontato più nel dettaglio i concetti legati agli ambienti, integrati nativamente in ASP.NET Core. Sempre in tema di ambienti ma sul lato infrastrutturale, abbiamo visto come sfruttare il web server Kestrel in configurazione di reverse proxy con altri web server tra cui Apache, NGINX e il classico IIS e, in particolare, abbiamo discusso di come questo porti innumerevoli vantaggi in termini di prestazioni, sicurezza e semplicità di gestione. Poiché a oggi i costi dell'hardware sono spesso un problema e considerando anche che la scalabilità è spesso imprevedibile, si è affrontato il tema cloud con Microsoft Azure e il servizio gestito degli App Service, che permettono di fare hosting e offrire scalabilità automatica alle applicazioni ASP.NET Core, mentre in seguito si è discusso di Docker, che va ad accrescere in modo esponenziale questi concetti, portando anche grosse semplificazioni nella fase di deployment, in cui i tempi sono decisamente più stretti.

Con questo ultimo capitolo il nostro viaggio alla scoperta di ASP.NET Core è terminato. In questi ventidue capitoli, abbiamo analizzato le nuove caratteristiche che ha portato, le peculiarità e i punti di forza di ASP.NET Core, analizzando in dettaglio la caratteristiche di ognuna delle sue funzionalità.

Vi abbiamo guidati attraverso un percorso che ha trattato le nozioni fondamentali, ponendo l'accento sulle novità e sulle caratteristiche più utili in un contesto in forte evoluzione, come quello rappresentato dal web. Speriamo di essere riusciti a trasmettervi almeno in parte la nostra passione e di avervi aiutato a costruire applicazioni web cross-platform, moderne e scalabili, basate su ASP.NET Core.

Buon lavoro e buon divertimento!

Informazioni sul Libro

Scritta per guidare gli sviluppatori alla scoperta di ASP.NET Core 2, il nuovo framework per il web cross platform e open source rilasciato da Microsoft, questa guida completa include tutte le ultime novità introdotte da ASP.NET Core e dalle tecnologie a corredo di applicazioni web, come Angular o l'accesso ai database.

Dalle basi di ASP.NET Core 2 ai concetti legati ad ASP.NET Core MVC, all'accesso ai dati, passando per identity e arrivando fino a JavaScript, Angular e tecnologie client-side, questo libro – con uno stile pratico e ricco di esempi – accompagna il lettore alla scoperta di tutte le caratteristiche che rendono ASP.NET Core uno dei toolkit più interessanti per sviluppare applicazioni web.

PUNTI DI FORZA

- Tutte le novità introdotte da ASP.NET Core
- I concetti base legati a ASP.NET Core MVC
- Gestione delle form con ASP.NET Core MVC
- Gestione dello stato
- Accesso ai dati con ADO.NET ed Entity Framework Core
- Sviluppo di servizi RESTful
- Gestione avanzata del runtime
- Globalizzazione e localizzazione
- Autenticazione e sicurezza delle applicazioni
- Sviluppo di applicazioni client-side e integrazione con ASP.NET Core